Efficient Index Structures for and Applications of the CompleteSearch Engine

Ingmar Weber

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der naturwisschaftlich-technischen Fakultäten
der Universität des Saarlandes

Saarbrücken
September, 2007

Tag des Kolloquiums: 16. November 2007
Dekan: Prof. Dr.-Ing. Thorsten Herfet

Prüfungsausschuss:
Vorsitz: Prof. Dr. rer. nat. Gert Smolka
Gutachter: Dr. Holger Bast und Prof. Dr.-Ing. Gerhard Weikum
Akadem. Beisitzer: Dr. Ernst Althaus

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Ort, Datum

(Unterschrift)

Kurzzusammenfassung

Typische Suchmaschinen, wie z.B. Google, erreichen Antwortzeiten deutlich unter einer Sekunden, selbst für einen Korpus mit mehr als einer Milliarde Dokumenten. Sie schaffen dies durch die Nutzung eines (parallelisierten) invertierten Index. Da der invertierte Index jedoch hauptsächlich für die Bearbeitung von einfachen Schlagwortsuchen konzipiert ist, bieten Suchmaschinen nur selten die Möglichkeit, komplexere Anfragen zu beantworten, die sich nicht in solch eine Schlagwortsuche umformulieren lassen, u.U. mit der Zurhilfenahme von speziellen Kunstworten.

Wir haben für die CompleteSearch Suchmaschine, konzipiert und implementiert am Max-Planck-Institut für Informatik, spezielle Datenstrukturen entwickelt, die ein deutlich größeres Spektrum an Anfragetypen unterstützen, ohne dabei die Effizienz zu opfern. Die CompleteSearch Suchmaschine baut auf einem kontext-sensitiven Präfixsuch- und Vervollständigungsmechanismus auf. Dieser Mechanismus ist einerseits einfach genug, um eine effiziente Implementierung zu erlauben, andererseits hinreichend mächtig, um die Bearbeitung zusätzlicher Anfragetypen zu erlauben.

Wir stellen zwei neue Datenstrukturen vor, die eingesetzt werden können, um das zu Grunde liegende Präfixsuch- und Vervollstängigungsproblem zu lösen. Die erste der beiden, AutoTree genannt, hat die theoretisch wünschenswerte Eigenschaft, dass sie für nicht entartete Korpora eine Bearbeitungszeit linear in der aufsummierten Größe der Ein- und Ausgabe zulässt. Die zweite, HYB genannt, ist auf die Komprimierbarkeit der Daten ausgelegt und ist für Szenarien optimiert, in denen der Index nicht in den Hauptspeicher passt, sondern auf der Festplatte ruht. Beide schlagen den Referenzalgorithmus, der den invertierten Index benutzt, um einen Faktor von 4-10 hinsichtlich der durchschnittlichen Bearbeitungszeit. Ein direkter Vergleich zeigt, dass im Allgemeinen HYB schneller ist als AutoTree.

Dank der HYB Datenstruktur kann die CompleteSearch Suchmaschine auch anspruchsvollere Anfragetypen, wie Facettensuche für Kategorieninformation, Vervollständigung zu Synonymen, Anfragen im Stile von elementaren, relationalen Datenbankanfragen und die Suche auf Ontologien, effizient bearbeiten. Für jede dieser Fähigkeiten beweisen wir die Realisierbarkeit unseres Ansatzes durch Experimente. Schließlich demonstrieren wir durch eine kleine Nutzerstudie mit Mitarbeitern des Helpdesks unseres Institutes auch den praktischen Nutzen unserer Arbeit.

Abstract

Traditional search engines, such as Google, offer response times well under one second, even for a corpus with more than a billion documents. They achieve this by making use of a (parallelized) inverted index. However, the inverted index is primarily designed to efficiently process simple key word queries, which is why search engines rarely offer support for queries which cannot be (re-)formulated in this manner, possibly using "special key words".

We have contrived data structures for the CompleteSearch engine, a search engine, developed at the Max-Planck Institute for Computer Science, which supports a far greater set of query types, without sacrificing the efficiency. It is built on top of a context-sensitive prefix search and completion mechanism. This mechanism is, on the one hand, simple enough to be efficiently realized by appropriate algorithms, and, on the other hand, powerful enough to be employed to support additional query types.

We present two new data structures, which can be used to solve the underlying prefix search and completion problem. The first one, called AutoTree, has the theoretically desirable property that, for non-degenerate corpora and queries, its running time is proportional to the sum of the sizes of the input and output. The second one, called HYB, focuses on compressibility of the data and is optimized for scenarios, where the index does not fit in main memory but resides on disk. Both beat the baseline algorithm, using an inverted index, by a factor of 4-10 in terms of average processing time. A direct head-to-head comparison shows that, in a general setting, HYB outperforms AutoTree.

Thanks to the HYB data structure, the CompleteSearch engine efficiently supports features such as faceted search for categorical information, completion to synonyms, support for basic database style queries on relational tables and the efficient search of ontologies. For each of these features, we demonstrate the viability of our approach through experiments. Finally, we also prove the practical relevance of our work through a small user study with employees of the helpdesk of our institute.

Zusammenfassung

Ist das “Suchproblem” denn noch nicht gelöst? Es gibt doch immerhin Google!

Wenn man sich bewusst macht, dass heutzutage kommerzielle Internetsuchmaschinen mehrere Milliarden von Webseiten in deutlich unter einer Sekunde durchsuchen können, um anschließend dem Anwender eine sortierte Liste mit (hoffentlichen) relevanten Dokumenten zu präsentieren, erscheint es vielleicht unklar, warum es lohnend sein könnte, an einer neuen Suchmaschinentechnologie zu arbeiten. Als Ausgangspunkt und Motivation für die in dieser Dissertation vorgestellte Arbeit ist es daher hilfreich, die Stärken und Schwächen eines Systems wie Google genauer zu betrachten.

Google[1] und ähnliche Internetsuchmaschinen[2] beeindrucken durch ihre unglaubliche Geschwindigkeit, mit der sie dem Anwender Suchergebnisse präsentieren. Nutzer haben verstanden, dass sie, solange sie ihr Informationsbedürfnis in klare, eindeutige Schlagworte fassen können, auf Google vertrauen können, wenn es darum geht (hoffentlich) relevante Dokumente in deutlich unter einer Sekunde zu finden. Google funktioniert großartig für solche Schlagwort-basierten Suchanfragen, denn dies ist genau die Anwendung, für die es konzipiert ist. Es gibt allerdings auch andere wünschenswerte Fähigkeiten, die man konzeptuell leicht einem Anwender bieten könnte.

Eine solche Fähigkeit ist *Präfixsuche*. Hier tippt der Nutzer lediglich die ersten paar Buchstaben eines Wortes und alle Dokumente mit einem Wort, das mit dieser Buchstabenfolge beginnt, werden dann für ihn gefunden. Dies erspart ihm das Tippen von weiteren Buchstaben, wenn der Präfix bereits hinreichend eindeutig ist (`greenp`[3]), es findet automatisch Wortvariationen mit verschiedenen Endungen (`demokra`[4]), und es gibt dem Anwender die Chance den Korpus zu “erkunden”, indem automatisch Worte für das selbe Konzept in Betracht gezogen werden (`pneumo`[5]).

Eine weitere Fähigkeit, die des Öfteren von Nutzen wäre, ist *Facettensuche*, bei der die Suchergebnisse in verschiedene Kategorien gruppiert werden, ähnlich wie man es von e-commerce Seiten wie ebay[6] kennt. Eine automatische Aufschlüsselung der Google Suchergebnisse (i) nach Sprache des Dokumentes, (ii) ob es eine private, wissenschaftliche oder kommerzielle Seite ist, oder (iii) nach Dateiformat, könnte den Filterprozess des Anwenders einfacher machen. Dadurch würde es auch überflüssig, manuell Kriterien für die “erweiterte Suche” angeben zu müssen, und dabei evtl. die Suchanfrage überzuspezifizieren, so dass man am Ende keinerlei Ergebnis bekommt.

Eine dritte, konzeptuell sehr einfache Fähigkeit wäre die Kombination von Informationen, die über verschiedene Dokumente verteilt sind. Z.B. erlaubt Google’s “Scholar” System[7] die Suche in wissenschaftlichen Arbeiten, wobei man sich dabei (in der “erweiterten Scholar-Suche”) auf bestimmte Autoren oder Konferenzen beschränken kann. Dennoch erlaubt es dem Nutzer nicht, in einer Anfrage nach allen Autoren zu fragen, die sowohl in der SIGIR als auch in der SODA Konferenz einen Beitrag veröffentlicht haben.

Der Grund für das Fehlen dieser Fähigkeiten ist derselbe Grund, der Google und ähnlichen Systemen ihre außerordentliche Performanz gibt: die Nutzung des invertierten Index. Alle großen Suchmaschinen basieren auf einem invertierten Index, der für jeden Term eine sortierte Liste von Dokumenten (oder genauer gesagt Dokumenten-Identifikationsnummern) bereithält. Der invertierte Index wird im nächsten Kapitel im Detail

[1] http://www.google.com
[2] http://www.live.com, http://search.yahoo.com
[3] Greenpeace.
[4] Demokratisch, Demokratie, Demokrat oder Demokraten.
[5] Präfix der Worte zum Thema Atmung und Lunge umfasst.
[6] http://www.ebay.com
[7] http://scholar.google.com

vorgestellt. Hier reicht es, sich einiger Charakteristika, die seinen Einsatz so attraktiv machen, bewusst zu sein: Das erste ist seine fast perfekte Zugriffslokalität, da die Bearbeitung dieser Listen normaler Weise ein lineares Durchgehen beinhaltet. Der zweite Vorteil besteht darin, dass diese Listen stark komprimierbar sind, was somit sowohl den Platzbedarf als auch die Lesezeit stark reduziert. Drittens ist es leicht möglich, einen invertierten Index über mehrere Maschinen zu verteilen. Die Aufteilung kann dabei sowohl nach Termen (jede Maschine enthält die Dokumentenlisten für ausgewählte Terme) wie nach Dokumenten (jede Maschine enthält die kompletten Informationen für bestimmte Dokumente) geschehen. Viertens kann ein invertierter Index effizient gebaut werden, selbst wenn die Daten nicht mehr in den Hauptspeicher passen. Dies geschieht mit Hilfe von für externen Speicher optimierten Sortieralgorithmen. Zusätzlich ist der invertierte Index durch das Hinzufügen von neuen Termen leicht erweiterbar.

Überraschender Weise ermöglicht der invertierte Index jedoch nicht die effiziente Bearbeitung von Anfragen der oben beschriebenen Typen. Hierfür gibt es hauptsächlich zwei Gründe: Erstens kann der invertierte Index nur (effizient) die Informationen für einzelne Terme bereitstellen. Aber z.B. braucht man sowohl für die Präfixsuche, für die eine alphabetische Folge von Worten relevant ist, als auch für die Facettensuche, wo die Menge der Kategoriennamen potentiell erheblich sein kann, die Informationen für eine (große) Menge von Worten, was für den invertierten Index ein Problem darstellt. Zweitens gibt eine Anfrage an einen invertierten Index "nur" Dokumente zurück. Um jedoch Autoren zu finden, die in zwei bestimmten Konferenzen etwas publiziert haben, muss man im Wesentlichen zwei Anfragen stellen, eine für jede Konferenz, und dann die Liste der Autoren (d.h. Terme) dieser Dokumente schneiden. Solch eine Operation (in der Datenbanksprache ein "Verbund" oder auf englisch "join" genannt) wird von Natur nicht ohne weiteres vom invertierten Index effizient unterstützt.

Wir haben Datenstrukturen für die CompleteSearch Suchmaschine entwickelt, die all diese Anfragetypen und noch weitere effizient unterstützen. Diese Datenstrukturen bieten eine effiziente Umsetzung eines einfachen aber dennoch mächtigen Mechanismus, der im nächsten Abschnitt informell vorgestellt wird, bevor er im nächsten Kapitel formal erfasst wird. Man beachte hierbei, das der Bezug dieses Mechanismus zu den drei oben besprochenen fehlenden Fähigkeiten nicht sofort offensichtlich ist. In der Tat besteht ein Beitrag dieser Arbeit darin, die Anwendbarkeit dieses Mechanismus für die Bereitstellung verschiedenere Fähigkeiten darzulegen.

Beschreibung des Kernmechanismus

Kontext-sensitive Autovervollständigungs-Suche bildet das Herzstück unserer CompleteSearch Suchmaschine. Autovervollständigung, in ihrer einfachsten Form, ist der folgende Mechanismus: Der Anwender tippt die ersten paar Buchstaben eines Wortes und dabei wird, entweder durch die Betätigung einer bestimmten Taste oder automatisch nach jedem Tastendruck, eine Methode aufgerufen, die alle Worte anzeigt, die Vervollständigungen der bisher getippten Buchstabenfolge sind. Dies hilft dem Anwender, mit möglichst geringem Aufwand schnell zu einer bestimmten Information zu navigieren, wobei auch nur ein teilweises Wissen (ein Präfix) des Zieles an sich benötigt wird. Das wohl bekannteste Beispiel dieses Mechanismus ist die Tabvervollständigung in der Unix Shell.

Das Problem, das wir in dieser Dissertation betrachten, beruht auf einer anspruchsvolleren Form der Autovervollständigung, die auch den *Kontext*, in dem das zu vervollständigende Wort getippt wurde, in Betracht zieht. Hier sollen (sofort nach jedem Tastendruck) nur die Vervollständigungen des letzten (teilweise) getippten Suchwortes angezeigt werden, die zu einem Treffer führen, also zu einem Dokument das alle (auch vorherige) Suchworte enthält. Man nehme zum Beispiel an, dass ein Anwender `information ret`[8] getippt hat. Vielversprechende Vervollständingungen könnten dann u.a. `retrieval` oder `return` sein, aber z.B. nicht, `retire`, da dies Wort an sich zwar vielleicht häufig vorkommt, die Kombination `information retire` aber nur zu wenigen (oder keinen) guten Treffern führt. Das zu Grunde liegende algorithmische Problem ist in Definition 1 im nächsten Kapitel formal erfasst. Diese Suchfähigkeit bezeichnen wir als *Autovervollständigungssuche* (da Autovervollständigung mit Suche kombiniert wird), oder auch, etwas länger aber dafür präziser, als *Präfixsuche mit Vervollständigung*.

[8] Man beachte, dass unser System echte Präfixsuche macht. D.h. alle Dokumente, die die Anfrage `information* ret*` erfüllen, werden als Treffer gewertet.

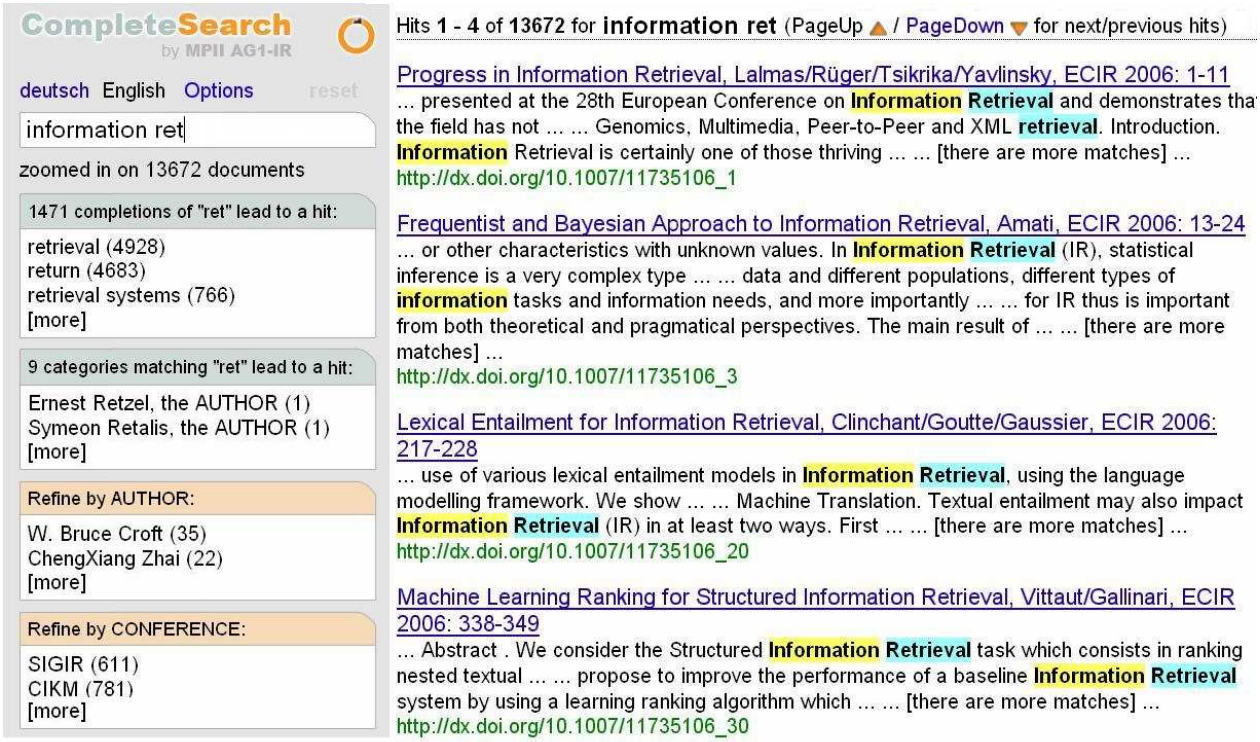

Figure 1: Bildschirmanzeige des Ergebnisses unserer Suchmaschine für die Anfrage `information ret`. Durchsucht wird eine Dokumentensammlung mit ungefähr 20.000 Publikationen aus dem Bereich Informatik, jede mit Volltext und Metadaten. Das Vervollständigungsfeld links und die Treffer auf der rechten Seite werden automatisch und ohne wesentliche Verzögerung nach jedem Tastendruck neu berechnet. Daher fehlt jede Art von Suchknopf völlig. Man beachte, dass die vorgeschlagenen Vervollständigungen neben normalen Worten ("return"), auch Phrasen ("retrieval system") und Kategoriennamen ("Ernest Retzel, the AUTHOR") beinhalten können. Die Zahl in Klammern hinter jeder Vervollständigung ist die Anzahl der Treffer, die man erhielte, wenn man diese Vervollständigung per Mausklick oder durch Eintippen auswählen würde. Allerdings besteht keinerlei Zwang, ein angefangenes Wort zu Ende zu tippen, da unsere Suchmaschine standardmäßig für alle Suchbegriffe eine Präfixsuche ausführt. Sollte der Anwender zum Beispiel ein neues Wort anfangen und `information ret data` tippen, so kämen die Vervollständigungen und Treffer für `data` (zum Beispiel `databases`) aus den 13.672 Treffern für `information ret`. Die unteren beiden Felder schlagen mögliche Verfeinerungen der Treffer durch Kategorieninformation vor, sofern diese Information zum Index hinzugefügt wurde. Dies ist die Facettensuche, die in Kapitel 6 genauer beschrieben wird.

Abbildung 1 zeigt die Bildschirmanzeige unserer CompleteSearch Suchmaschine mit dem Ergebnis für die Anfrage `information ret`. Eine List mit online verfügbaren Demonstratoren der Suchmaschine für verschiedene Dokumentensammlungen findet sich unter `http://search.mpi-inf.mpg.de/`. Die zusätzlichen Suchfähigkeiten, wie z.B. die in der Abbildung erkennbare Facettensuche, so wie weitere Fähigkeiten die im nächsten Abschnitt erwähnt und in späteren Kapiteln detailliert erörtert werden, lassen sich alle (effizient) durch ein und denselben Mechanismus realisieren.

Wissenschaftlicher Beitrag und Inhaltsübersicht

Die grobe Übersicht dieser Dissertation ist einfach: Zuerst geben wir die formale Problemdefinition und diskutieren disbezüglich relevante Arbeiten. Dann stellen wir unsere Algorithmen zur Lösung des Problems vor. Anschließend präsentieren wir verschiedene Erweiterungen und Anwendungen des zu Grunde liegenden Mechanismus, bevor wir, vor dem Fazit, schließlich noch einige wichtige Implementierungsaspekte betrachten. Es folgt eine detailliertere Inhaltsaufschlüsselung und kurze Zusammenfassung unseres Beitrages für jedes Kapitel.

In Kapitel 2 formalisieren wir das algorithmische Problem, welches im Zentrum unserer Suchmaschine steht. Ferner zeigen wir, wie die gängigste Datenstruktur im Bereich Information Retrieval, der invertierte Index, zur Lösung dieses Problemes eingesetzt werden kann. Hierfür geben wir eine theoretische Analyse seiner Laufzeit und zeigen, wo seine Schwächen liegen. Der invertierte Index ist die Referenzdatenstruktur in dieser Dissertation, und wir vergleichen unsere Datenstrukturen dagegen. Wir diskutieren auch die Anwendbarkeit anderer existierender Datenstrukturen für unser Problem, insbesondere die von Suffixarrays.

In Kapitel 3 stellen wir unsere erste Datenstruktur (AutoTree) vor. Wir beweisen sowohl theoretisch, unter milden Bedingungen, und experimentell, dass seine Laufzeit Ausgabe-abhängig (output-sensitive) ist. D.h. die Laufzeit des Algorithmus ist proportional zur Ergebnismenge. Wir zeigen experimentell dass AutoTree für eine große Klasse von Anfragen deutlich kürzere Antwortzeiten als der invertierte Index bietet. Dieses Kapitel beruht auf gemeinsamer Arbeit mit Holger Bast und Christian Worm Mortenssen und wurde in einer vorläufigen Fassung in der Konferenz SPIRE 2006 (13th International Conference on String Processing and Information Retrieval) [Bast 06b] präsentiert.

Im darauf folgenden Kapitel 4 wird unsere zweite Datenstruktur (HYB) vorgestellt. HYB ist bezüglich I/O (Eingabe/Ausgabe) Performanz optimiert und sein Platzverbrauch kommt nahe an die theoretische untere Schranke der Entropie heran. Obwohl HYB sogar eine gewisse Mindestlaufzeit hat, schlägt er den invertierten Index für nicht-entartete Anfragen. Wir vergleichen auch AutoTree und HYB experimentell miteinander und zeigen, dass im Allgemeinen HYB mit seiner Zugriffslokalität vorzuziehen ist. Der Großteil dieses Kapitels wurde im Konferenzband von SIGIR 2006 (29th International Conference on Research and Development in Information Retrieval) veröffentlicht [Bast 06c] und ist gemeinsame Arbeit mit Holger Bast.

Während die Kapitel 2-4 sich auf eine effiziente Umsetzung der Kernfähigkeit konzentrieren, wird in Kapitel 5 zunächst erneut der Nutzen der Autovervollständigungssuche erörtert, bevor wir unser System mit verschiedenen anderen Systemen vergleichen, die jeweils ähnliche Suchmöglichkeiten wie die CompleteSearch Suchmaschine bieten. In diesem Kapitel diskutieren wir auch einige einfache Erweiterungen des grundlegenden Autovervollständigungsmechanismus, die den Nutzen der Grundfähigkeit weiter erhöhen. Dies sind: Relevanzsortierung der Trefferdokumente und Vervollständigungen, Nähesuche, Bearbeitung von ODER und NICHT Anfragen, Teilwortsuche und Autovervollständigung zu Phrasen. Während die Erweiterungen in der obigen Liste in keiner Weise an den Präfixsuchmechanismus gebunden sind, bedürfen die Erweiterungen, die in den anschließenden Kapiteln 6-9 präsentiert werden, einer effizienten Implementierung unseres Kernmechanismus.

In der Facettensuche wird die Navigation in Verzeichnissen, für Dokumentensammlungen die gemäß verschiedenen Kategorien klassifiziert sind, kombiniert mit normaler Schlagwortsuche. Im Kapitel 6 zeigen wir, wie man unsere Arbeit leicht anwenden kann, um effiziente Facettensuchfähigkeiten zu erhalten. Dies ist eine Zusammenarbeit mit Holger Bast und wurde in einem Workshop über Facettensuche bei der SIGIR 2006 vorgestellt [Bast 06d]. Unseres Wissens nach war dies das erste Mal, dass statt der Nutzbarkeit der Effizienzaspekt der Facettensuche untersucht wurde.

In Kapitel 7 erweitern wir den Autovervollständigungsmechanismus so, dass nicht nur Vervollständigungen eines Präfixes sondern auch verwandte Terme oder Synonyme vorgeschlagen werden. Wir zeigen wie man, sofern man Gruppen von verwandten Termen oder Synonymen kennt, (i) dieses Wissen ausnutzen kann, um für einen bestimmten Anfragekontext diese Vorschläge effizient zu erhalten und (ii) wie man dabei eine übermäßige Vergrößerung des Indexes verhindern kann. Dieses Kapitel basiert auf gemeinsamer Arbeit mit Holger Bast und Debapriyo Majumdar und wird bei CIKM 2007 (16th Conference on Information and Knowledge Management) [Bast 07b] vorgestellt.

In Kapitel 8 zeigen wir, wie die CompleteSearch Suchmaschine mit ihrer effizienten Präfixsuche und, wie sich zeigen wird, Verbundberechnung (englisch: “join”) benutzt werde kann, um eine Mischung aus Datenbankanfragen (“Welche Autoren haben sowohl in SIGIR wie auch in SODA veröffentlicht?”) und Volltextsuchanfragen (“Finde alle Veröffentlichungen, die sowohl die Worte ‘Datenbank’ wie auch ‘Relevanzsortierung’ enthalten.”) zu bearbeiten. Dadurch wird zumindest teilweise eine Brücke zwischen klassischen Datenbanksystemen und Suchmaschinen geschlagen. Der Inhalt diese Kapitels entstand in Zusammenarbeit mit Holger Bast und wurde zum Großteil im Konferenzband von CIDR 2007 (Third Biennial Conference on Innovative Data Systems Research) veröffentlicht [Bast 07c].

Kapitel 9 baut stark auf den Ideen des vorangehenden Kapitels auf und erweitert diese noch. In diesem Kapitel zeigen wir, wie man die CompleteSearch Suchmaschine zu einer semantischen Suchmaschine erweitern

kann. Sie ist in dem Sinne “semantisch”, als sie ontologisches Wissen nutzt, um das Suchen nach Entitäten mit bestimmten Eigenschaften, z.B. Personen die in einem bestimmten Jahr geboren oder Mitglieder einer bestimmten Gruppe sind, zu ermöglichen. Dies Kapitel beruht auf einer Zusammenarbeit mit Holger Bast, Alexandru Chitea und Fabian Suchanek. Es wurde im Konferenzband von SIGIR 2007 (30th International Conference on Research and Development in Information Retrieval) [Bast 07a] veröffentlicht.

Der Ausgangspunkt unserer Arbeit war der Glaube (oder damals eher die Hoffnung), dass unser System für den Nutzer einen spührbaren Mehrwert darstellen würde. Wir haben eine kleine Nutzerstudie mit Angestellten des Helpdesks unseres Institutes durchgeführt, um diesen Glauben zu überprüfen. Diese Nutzerstudie wird in Kapitel 10 vorgestellt und ihre (ermutigenden) Ergebnisse wurden im Konferenzband von GWEM 2007 (German Workshop on Experience Management) [Bast 07d] veröffentlicht. Dieser Workshop fand in Verbindung mit der vierten Konferenz Professionelles Wissensmanagement (WM 2007) statt.

Eine Reihe von wichtigen Implementierungs- und Designentscheidungen werden in Kapitel 11 erörtert. Diese sind teilweise von einer Art, wo sie für die effiziente Bearbeitung von Anfragen relevant sind, und teilweise von einer Art, wo sie die einfache Ergänzungen von neuen Suchmöglichkeiten für unser System ermöglich(t)en.

Nur wenig Hintergrundwissen des nun folgenden Kapitels 2, insbesondere jedoch Definition 1, wird in späteren Kapiteln vorausgesetzt (oder ist dort zumindest nützlich). Davon abgesehen sind alle Kapitel in sich selbst abgeschlossen und enthalten, wo dies Sinn macht, einen eigenen Abschnitt mit experimenteller Evaluierung. Experimente für die erweiterten Suchmöglichkeiten (Kapitel 6 - 10) wurden nur mit der HYB Datenstruktur gemacht, da sie sich im Allgemeinen als die bessere Datenstruktur erwies (siehe Abschnitt 4.6) und das Herzstück unserer CompleteSearch Suchmaschine bildet.

Acknowledgments

I'm not sure, what state this dissertation would be in now, were it not for the reliable supervision, support and help of my supervisor Holger Bast. I undoubtedly profited greatly from his guidance in every aspect of my work. His devotion to students at any level, his attempt to always bridge the gap between theory and practice and his talent for giving accessible scientific talks were truly inspirational, and I can only hope, that I'll be able to pass on a bit of this inspiration in the future. Thank you Holger!!

Thanks go also to all the people at the MPI, who help in their own way to create a friendly and open atmosphere; to everyone, whom I could motivate during the last years to donate to a charity; to all my former colleagues, for joining the 12:30 lunch group, for joining various activities, for having random conversations, and for always having their doors open; to the roughly 30 volunteers, who contributed to the "Cool Stuff on the Web" seminar; and to everyone, who made my time in Saarbrücken much richer in many different ways and overall simply more enjoyable; in particular *Andreas*, *Barbara*, *Dina G.*, *Gernot*, *Ina*, *Irina*, *Juliane*, *Khaled*, *Petr*, *Ralitsa*, *Susi*, *Waqar* and *Will*.

Special thanks go to all my former roommates (*Christina*, *Karine*, *Mona*, *Carole*, *Sebastian*, *Dina H.* and *Aishu*, for putting up with all my bad habits, for being there to talk to and to listen after a long day, and for making me feel at home; to *Christian*, for the numerous times he helped me with all sorts of basic Linux problems (without laughing too loud), for putting up with my mess, and for being a fantastic office mate; to *Ralf*, for co-organizing the best parties in Saarbrücken, for ensuring a continued supply of our research group with food and drinks, and for not being too grown-up for dancing with me in the MPI; to *Katja*, for having a great sense of humor, for uncountable dances in the Havanna Club, and for being a wonderfully unconventional person; to the members of the *Hospitality Club*, for spreading a bit of international friendship around the globe; and to *Milka* and *Rittersport*, for providing a delicious 100-gr breakfast, lunch and dinner.

Last but not least, I'd like to thank my mother for her perpetual support of anything I do.

Contents

Chapter 1

Introduction

1.1 Is the "search problem" not solved? I mean, there's Google!

Given that nowadays commercial internet search engines can search through several billions of web documents in well under one second, and then present the user with a ranked list of (hopefully) relevant documents, it might not be clear, why it could be fruitful to work on a new search engine technology. As a starting point and motivation for the work in this dissertation, it is helpful to ponder for a moment the strengths and weaknesses of a Google-like system.

Google[1] and similar web search engines[2] impress by the blazing speed, with which they present results to the user. Users have learned that, as long as they can phrase their information need in terms of unambiguous, unique key words, they can rely on Google to provide them with a set of (hopefully) relevant documents in well under one second. Google works great for such key word based retrieval, because this is exactly what it is built for. There are, however, other desirable search features, which would conceptually be easy to offer to the users.

One such feature is *prefix search*, where the user only enters the first few letters of a word and all documents containing a word starting with this sequence are retrieved. This features saves typing, when a prefix (`greenp`[3]) is already discriminative enough, it will automatically retrieve word variations with different endings (`democra`[4]), and it gives the user a chance to "explore" the corpus by automatically including other words for the same concept (`pneumo`[5]).

Another often desirable feature is *faceted search*, where the search results are grouped into different categories, similar to what is done on e-commerce sites such as ebay[6]. An automatic breakdown of the Google search results according to (i) the document language, (ii) whether it comes from more of a private, scientific or a commercial site, or (iii) the file format, could make the result filtering process by the user easier, while removing the burden of having to specify "advanced search" options, possibly over-specifying the result requirements.

A third conceptually easy feature involves the combination of information, spread across several documents. For example, Google's "scholar" search[7] offers a search of scientific documents refined (in the "advanced search" options) by author or by conference. Yet it does not allow the user to pose a query asking for all authors who have published in both the SIGIR and the SODA conference.

The reason that such features are not supported is the same reason that gives Google and similar systems their extraordinary performance: the use of the inverted index. All major search engines are based on an inverted index, which precomputes for every term a sorted list of all documents (or rather their ids) containing the term. The inverted index will be discussed in more detail in the next chapter, but for now it suffices to note some of its characteristics, which make it so attractive to use. The first is that it has an almost perfect locality of access, as handling these lists usually involves linear scans. The second advantage is that these lists are highly compressible, vastly reducing the amount of space needed to store them and the time to read them.

[1] http://www.google.com
[2] http://www.live.com, http://search.yahoo.com
[3] Greenpeace.
[4] Democratic, democracy, democrat or democrats.
[5] Prefix pertaining to breathing, respiration and the lungs.
[6] http://www.ebay.com
[7] http://scholar.google.com

Thirdly, an inverted index can be easily distributed among multiple machines, both with respect to terms (where each machine holds the document lists for selected terms) and with respect to documents (where each machine is responsible for the data pertaining to certain documents). Fourthly, it can be efficiently constructed, even when the data no longer fits in main memory, using external memory sorting routines. Finally, it can be easily extended by adding new terms to the index.

Somewhat surprisingly, the inverted index does not allow the processing of queries of the types mentioned above in an efficient manner. The reason for this is essentially two-fold: On the one hand, the inverted index can only (efficiently) provide information about individual terms. But, e.g., in the cases of prefix search, where a range of words is of relevance, or for faceted search, where the set of labels for directories could be potentially considerable, information about a (large) set of words is required, which poses a problem for the inverted index. On the other hand, the inverted index returns "only" documents. But to find authors who have published in two given conferences, we essentially need to retrieve documents for two queries, one for each conference, and then intersect the lists of authors (i.e., terms) for these documents. Such an operation (a database "join") is not inherently supported by an inverted index.

We have developed data structures for the CompleteSearch engine, which efficiently provide all of the features mentioned above, as well as several others. These data structures offer an efficient realization of a simple, yet powerful mechanism, which will be introduced informally in the next section, before it is formalized in the next chapter. Note that the applicability of this mechanism to the three features missing in Google, mentioned above, will not be immediately obvious. Indeed, showing the connection between this mechanism and various features is one of the contributions of this work and the relation will become clear in later chapters.

1.2 Description of the Core Mechanism

A context-sensitive autocompletion search is at the heart of our CompleteSearch engine. Autocompletion, in its most basic form, is the following mechanism: the user types the first few letters of some word, and either by pressing a dedicated key or automatically after each keystroke a procedure is invoked that displays all relevant words that are continuations of the typed sequence. This helps the user to navigate to a desired piece of information quickly and with as little effort as possible and only requires partial knowledge (a prefix) of the information itself. The most prominent example of this feature is the tab-completion mechanism in a Unix shell.

The problem we address in this dissertation, is derived from a more sophisticated form of autocompletion, which takes into account the *context* in which the to-be-completed word has been typed. Here, we would like an (instant) display of only those completions of the last query word which lead to hits, i.e., documents containing all the entered query words, as well as a display of such hits. For example, assume a user has typed `information ret`[8]. Promising completions might then be `retrieval`, `return`, etc., but not, for example, `retire`, assuming that, although `retire` by itself is a frequent word, the query `information retire` leads to only a few good hits. The underlying algorithmic problem is formalized in Definition 1 in the next Chapter. This is the feature we refer to as *autocompletion search* (as it combines autocompletion with search) or, more concretely but somewhat less concisely, as *prefix search and completion*.

Figure 1.1 shows a screenshot of our CompleteSearch engine responding to the query `information ret`. For a list of available live demos, see `http://search.mpi-inf.mpg.de/`. The additional features, such as faceted search, which can also be seen in the screenshot, and others mentioned in the next section and discussed in detail in later chapters, can all be (efficiently) supported via the same mechanism.

1.3 Contributions and Outline

The rough outline is simple: first the formal problem definition and related work, then our algorithms for its solutions, followed by various extensions and applications of the basic mechanism, and, before the conclusions, finishing with important implementation aspects of the CompleteSearch engine, which we have built. A more detailed chapter-by-chapter breakdown, with a short summary of the contributions, follows.

[8] Observe that our system does full prefix search. So any document matching `information* ret*` would be returned as a hit.

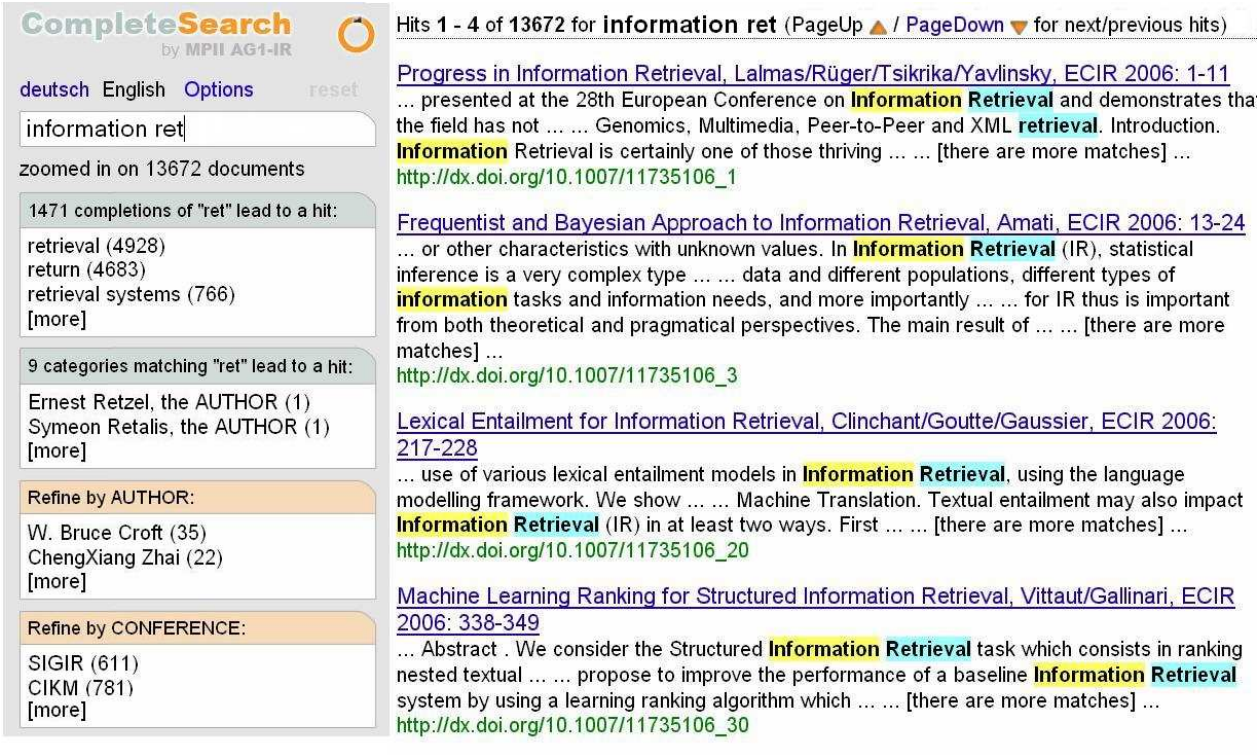

Figure 1.1: A screenshot of our search engine for the query `information ret` searching in a collection of about 20,000 computer science articles, each with full text and meta data. The completion boxes on the left and the hits on the right are updated automatically and instantly after each keystroke, hence the absence of any kind of search button. Note that the suggested completions can be words ("return"), phrases ("retrieval system"), and category names ("Ernest Retzel, the AUTHOR"). The number in parentheses after each completion is the number of hits that would be obtained if that completion was selected or typed. Query words need not be completed, however, because the search engine, by default, does an implicit prefix search on all query words. If, for example, the user continued typing `information ret data`, completions and hits for `data` (for example, `databases`, would be from the 13,672 hits for `information ret`. The two lower boxes suggest possible refinements of these hits via whatever category information was added to the index. This is the faceted search feature described in Chapter 6.

In Chapter 2, we formalize the algorithmic problem that is at the heart of our engine. We also show how an inverted index, the standard data structure in information retrieval, can be used to solve this problem, we given a theoretical analysis of its running time and show where its shortcomings lie. The inverted index will be the baseline algorithm throughout this dissertation, and we will compare our data structures against it. We also discuss various other existing data structures, in particular suffix arrays, that might be used to tackle the problem.

In Chapter 3, we present the first data structure developed by us, called AutoTree. We prove both theoretically, under mild assumptions, and experimentally that its running time is output-sensitive, i.e., the algorithm takes time proportional to the size of the output. We demonstrate through experiments that AutoTree outperforms the inverted index on a wide range of inputs. This chapter is based on joint work with Holger Bast and Christian Worm Mortenssen and was presented in preliminary form in the 13th International Conference on String Processing and Information Retrieval (SPIRE 2006) [Bast 06b].

In the following Chapter 4, our second data structure HYB is introduced, which is optimized for I/O performance and whose space consumption gets close to theoretical lower bounds derived from the entropy. Although HYB actually has a certain minimal running time, it beats the inverted index for general inputs. We also give an experimental comparison of AutoTree vs. HYB to show that in most settings HYB, with its locality of access is preferable in practice. Most of this chapter, was published in the proceedings of the 29th International Conference on Research and Development in Information Retrieval (SIGIR 2006) [Bast 06c] and is joint work with Holger Bast.

Whereas Chapters 2-4 focus on efficient realizations of the feature, Chapter 5 assesses the usefulness of autocompletion search again, before comparing several other systems, each providing a service similar to our CompleteSearch engine. In that chapter, we also discuss various simple extensions of the basic autocompletion mechanism, which add to its usefulness. These are the ranking of the matching results and completions, proximity search, OR and NOT queries, subword search and autocompletion to phrases. Whereas the extensions listed above are not prefix search specific but are independent of this, the ones presented in the then following Chapters 6-9 heavily depend on an efficient realization of our central mechanism.

In faceted search, directory browsing is combined with key word based search, for document collections which are organized by various categories. In Chapter 6, we show how to apply our work to easily obtain efficient faceted search capabilities. This is joint work with Holger Bast and was presented at the Workshop on Faceted Search at SIGIR 2006 [Bast 06d]. To our knowledge, this was the first time that the efficiency aspect, rather than the usability aspect, of faceted search was studied.

In Chapter 7, we extend our autocompletion mechanism from suggesting only completions for a prefix to also suggest related terms or synonyms. We show how, given sets of related terms or synonyms, we can harvest this information in such a way that we can (i) find, for the query context given, these suggestions efficiently, and (ii) we do not inadequately increase the size of the index doing this. The work in this chapter is joint work with Holger Bast and Debapriyo Majumdar, and will be presented at the 16th Conference on Information and Knowledge Management (CIKM 2007) [Bast 07b].

In Chapter 8, we demonstrate how with its efficient prefix search and, as we will show, join mechanism the CompleteSearch engine can be used, to answer a mix of classical db-style (“Which authors have published both in SIGIR and SODA?”) and full-text queries (“List all publications containing the words database and ranking.”), partly bridging the gap between DB and IR systems. Most of Chapter 8 is work published in the proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR 2007) [Bast 07c] and is joint work with Holger Bast.

Chapter 9 heavily builds upon and extends the ideas from the previous chapter. Here we show how to incorporate our CompleteSearch engine into a semantic search engine. It is “semantic” in the sense that it (efficiently) uses ontological knowledge to allow searching for entities with certain properties, e.g., people born in a given year or members of a given group. This is joint work with Holger Bast, Alexandru Chitea and Fabian Suchanek. It was published in the proceedings of the 30th International Conference on Research and Development in Information Retrieval (SIGIR 2007) [Bast 07a].

The initial starting point for our work was the belief (or at that time rather the hope) that our system would give a noticeable added value to the user. We conducted a small user study with employees from our institute’s helpdesk to (successfully) verify this belief. This user study is presented in Chapter 10 and its (encouraging) results, again joint work with Holger Bast, were published in the proceedings of the German Workshop on

Experience Management (GWEM 2007) [Bast 07d], which was held in conjunction with the Vierte Konferenz Professionelles Wissensmanagement (WM 2007).

A number of important implementation and design choices are discussed in Chapter 11. These are partly of a nature, where they are relevant for efficient query processing, and partly of a nature, where they allow(ed) a simple addition of new features to the system.

Only some background from the now following Chapter 2, in particular Definition 1, is assumed (or at least helpful) in the later chapters. Apart from this, all chapters are self-contained and, where applicable, contain a section with experimental evaluation. Experiments for the advanced features and extensions (Chapters 6 - 10) are for the HYB index only, as it came out as the preferable data structure in a general setting, see Section 4.6, and as it is at the heart of our CompleteSearch engine.

Chapter 2

Problem Definition and Baseline Algorithm

In the following Section 2.1, we formalize the aforementioned autocompletion search mechanism. The then following Section 2.2 gives further insights related to the difficulty of the problem by presenting a first concrete solution. This solution uses inverted lists and serves as a baseline throughout this dissertation. The last but one Section 2.3 discusses other algorithms, which could be applied to our algorithmic problem, most notably suffix arrays. Finally, Section 2.4 summarizes the (little) notation used in this and the remaining chapters, which is mostly given for reference purposes.

2.1 Formal Problem Definition

The following problem definition formalizes the algorithmic problem underlying the autocompletion search feature, described informally in Section 1.2. Chapters 3 and 4 then present our AutoTree and HYB data structures, which can be used to solve this problem.

Definition 1 *An* autocompletion search query *is a pair* (D, W)*, where* W *is a range of words (all possible completions of the last word which the user has started typing), and* D *is a set of documents (the hits for the preceding part of the query). To process the query means to compute the set* Φ *of all word-in-document pairs* (w, d) *with* $w \in W$ *and* $d \in D$*, as well as both the set of matching documents* $D' = \{d : \exists (w, d) \in \Phi\}$ *and the set of matching words* $W' = \{w : \exists (w, d) \in \Phi\}$*. (Note that for the very first query word,* D *is the set of all documents.)*

Remark. Algorithms based on the intersection of (sorted) lists of document ids, such as INV, discussed in this chapter, or HYB, discussed in Chapter 4, will require both the input set D and the output set D' to be *sorted* sequences, rather than an unsorted sets. Our AutoTree algorithm, discussed in the following Chapter 3, does not require this additional property.

Given an algorithm for solving autocompletion queries according to the definition above, we obtain the desired search feature from Section 1.2 as follows: For the example query `information ret`, W would be all words from the vocabulary starting with `ret`, and D would be the set of all hits for the query `information`. The output Φ would be all word-in-document pairs (w, d), where w starts with `ret` and d contains w as well as a word starting with `information`, [1] D' would be all such documents d and W' the corresponding union of words w.

Now if the user continues with the last query word, e.g., `information retri`, the set of candidate documents D does not change. This allows us to simply filter the sequence of word-in-document pairs from the previous query (with the same D but a larger W), keeping only those pairs (w', d'), where w' starts with `retri`. This will, in practice, always be faster than relaunching a full autocompletion search query. Note that this filtering is independent of the method used to compute the initial result. See Section 11.8 for details on this and other uses of result caching.

If, on the other hand, the user starts a new query word, e.g., `information ret meth`, then we have another autocompletion query according to Definition 1, where now W is the set of all words from the vocabulary

[1] We always assume an implicit prefix search, that is, we are actually interested in hits for all words *starting* with `information`, which is usually what one wants in practice. Whole-word-only matching can be enforced by introducing a special end of word symbol `$`.

starting with `meth`, and D is the set of all hits for `information ret`. In a general setting, this set of the new candidate documents can be obtained from the sequence of matching word-in-document pairs for the last query by sorting the matching (w, d) pairs according to d. This sort takes time $O((\sum_{w\in W} |D\cap D_w|)\log(\sum_{w\in W} |D\cap D_w|))$ and, while finding the unique elements, also guarantees that the elements of D will be sorted. However, both of our algorithms, presented in the next two chapters, manage to avoid this cost, by either not requiring the elements of D to be sorted (as is the case for AutoTree), or by working with blocks already sorted by document id (as is the case for HYB). Details are given in the respective chapters.

In practice, we are actually interested in the *best* hits and completions for a query. This can be achieved by following standard approaches and is discussed in detail in Section 5.4. In fact, the main reason that we chose all matching (w, d) pairs Φ to be part of the output in Definition 1, rather than only the matching documents D' and words W', is that we will usually require *all* matching pairs to give an appropriate ranking.

2.2 Using the Inverted Index to Answer Autocompletion Search Queries

In this section, we will first define what we mean by an "inverted index". Then we will analyze its space consumption, before we show how to answer an autocompletion search query (Definition 1) using an inverted index. This will be the main focus of this section. It will be done through a formal analysis of its processing time for such queries, presenting upper and lower bounds, as well as an average case analysis. Extensions, such as compression of the index and the incorporation of positional information, will be discussed in later chapters, when the machinery required for the corresponding analysis has been set up. The aim of this section is (i) to provide the reader with more intuition concerning the problem itself, and (ii) to give us a baseline against which we will compare our data structures both theoretically and through experiments.

2.2.1 The Inverted Index: Definition and Space Analysis

Definition 2 *By* INV *(inverted index) we mean the following data structure:*
For each word w, store the list of all (ids of) documents D_w containing that word, sorted in ascending order.

The elements of the inverted lists, are just a rearrangement of the sets of all word-in-document pairs. The cardinality of this set, which is essentially the size of the corpus, we denote by N. Each document id can be encoded with $\lceil \log_2 n \rceil$ bits. So the total (uncompressed) space usage is given by the following lemma. Space for storing the lengths of the lists is not included in this bound.

Lemma 1 *The inverted lists for INV can be stored uncompressed using a total of at most $N \cdot \lceil \log_2 n \rceil$ bits.*

INV's intrinsic space efficiency (the "entropy") and its compressibility will be discussed in Section 4.3.1, once the required terms and concepts have been introduced. Compression will not change its asymptotic processing time, which is discussed in the following.

2.2.2 INV's Processing Time

In the rest of this section, we analyze the time complexity of processing autocompletion search queries with INV and point out two inherent problems at the end of this section.

Lemma 2 *With INV, an autocompletion query (D, W) can be processed in the following time, where D_w denotes the inverted list for word w:*

$$|D| \cdot |W| + \sum_{w\in W} |D_w| + \sum_{w\in W} |D \cap D_w| \cdot \log |W|.$$

Assuming that the elements of W, D, and the D_w are picked uniformly at random from the set of m words and the set of n documents, respectively, this bound has an expected value of

$$|D| \cdot |W| + \frac{|W|}{m} \cdot N + \frac{|D|}{n} \cdot \frac{|W|}{m} \cdot N \cdot \log |W|.$$

INV's processing time is bounded below by $\Omega(\sum_{w\in W} \min\{|D|, |D_w|\})$.

Remark. By picking the elements of a set S at random from a superset U, we mean that each subset of U of size $|S|$ is equally likely for S. We are *not* making any randomness assumption on the *sizes* of W, D, and D_w above.

Proof. The obvious way to use an inverted index to process an autocompletion query (D, W) is to compute, for each $w \in W$, the intersections $D \cap D_w$. Then, W' is simply the set of all w for which the intersection was non-empty, and D' is the union of all (non-empty) intersections and, to obtain Φ, for each element in $d \in D \cap D_w$ we add (w, d) to the output. The intersections can be computed in time linear[2] in the total input volume $\sum_{w \in W}(|D| + |D_w|)$. The union D' can be computed by a $|W|$-way merge, which requires on the order of $\log |W|$ time per element scanned. Note that the total sum of the lengths $|D_w|$ over all m words in the vocabulary is N, which is the total number of word-in-document pairs. With the randomness assumptions, the expected size of a single list D_w is thus N/m. Assuming that the elements of both D and D_w are picked uniformly at random from the set of all documents of size n, the expected size of the intersection $|D \cap D_w|$ is $|D|/n \cdot N/m$, as the probability that a certain element is contained in both sets is $|D|/n \cdot N/(mn)$. For the lower bound, observe that INV computes one intersection for each $w \in W$ and any algorithm for intersecting D and D_w has to differentiate between $2^{\min\{|D|,|D_w|\}}$ possible outputs. Assuming it is a comparison-based intersection algorithm, it will for a general input need at least $\min\{|D|, |D_w|\}$ comparisons.[3]

■

Lemma 2 highlights two problems of INV. The first is that the term $|D| \cdot |W|$ can become prohibitively large: in the worst case, when D is on the order of n (i.e., the first part of the query is not very discriminative) and W is on the order of m (i.e., only few letters of the last query word have been typed), the bound is on the order of $n \cdot m$, that is, quadratic in the collection size. The second problem is due to the required merging. While the volume $\sum_{w \in W} |D \cap D_w|$ will typically be small once the first query word has been completed, it will be large for the first query word, especially when only few letters have been typed. As we will see in Sections 3.6 and 4.5, INV frequently takes seconds for some queries, which is quite undesirable in an interactive setting. This is exactly what motivated us to develop more efficient index data structures.

Note that both problems ultimately arise because INV does not exploit the fact the elements in W form a *range*. The very same running times could be obtained if the elements in W were arbitrary elements.

2.3 Related Work

This section discusses work related to the general *algorithmic problem* given in Definition 1, as the following two chapters will focus on data structures, which can be used to solve this problem. Aspects related to (i) the usability of the corresponding *autocompletion search feature* or (ii) a particular feature (such as faceted search), which hinges on an efficient solution to the algorithmic problem of Definition 1, are *not* discussed here but in the corresponding chapters.

There is a large variety of alternatives to the inverted index in the literature. The ones that apply to the autocompletion search problem are discussed here. One of the most straightforward ways to process an autocompletion search query (D, W), would be to explicitly search each document from D for occurrences of a word from W. However, this document-by-document approach has a very poor locality of access and would give us a non-constant query processing time per element of D, completely independent of the respective $|W|$ or output size $\sum_{w \in W} |D \cap D_w|$. For these reasons, we do not consider this approach further in this work. Another approach would be to use *Signature files*, which store supersets of items in a manner similar to bloom filters. However, in [Zobel 98] they were found to be in no way superior to (but significantly more complicated than) the inverted index in all major respects.

Our autocompletion problem is related to, but distinctly different from *multi-dimensional range searching problems*, where the collection consists of tuples (of some fixed dimension, for example, pairs of word prefixes), and queries are asking for all tuples that match a given tuple of ranges [Gaede 98; Arge 99; Ferragina 03; Alstrup 00]. These data structures could be used for our autocompletion search problem, provided that we were willing to limit the number of query words. For fast processing times, however, the space consumption

[2] There are asymptotically faster algorithms for the intersection of two lists [Baeza-Yates 04; Demaine 00], but in our experiments, we got the best results with the simple linear-time intersect, which we attribute to its compact code and perfect locality of access.

[3] We don't know of any intersection algorithm that is (i) *not* comparison-based and (ii) does not need to scan either of the two input lists completely.

of any of these structures would be on the order of N^{1+d}, where N is the size of an inverted index, and $d > 0$ grows (fast) with the dimension. For our autocompletion search queries, we can achieve fast query processing times and space efficiency at the same time because we have the set of documents matching the part of the query before the last word already computed (namely when this part was being typed). In a sense, our autocompletion problem is therefore a 1 1/2 - dimensional range searching problem.

When searching for prefixes (or arbitrary patterns) in a text collection, suffix arrays are a standard choice [Manber 90; Grossi 00; Grossi 04]. Although these approaches are not directly applicable to our autocompletion problem, we could indeed use suffix arrays to produce the list of all documents that contain words with a given prefix (or even infix). This list could then be intersected with the set D.

The reason why we have taken INV as our baseline, and not an algorithm based on suffix arrays, as just outlined, is as follows. Uncompressed suffix arrays use too much space, as they index every character of the collection.[4] Compressed suffix arrays are not competitive with respect to running time when it comes to *reporting* and not just *counting* the occurrences of an infix, because each reported occurrence requires a large number (depending on the compression ratio) of operations and typically incurs at least one cache miss.

Note that the situation would (seem to) be different, if we wanted context-sensitive infix search. Suffix arrays would give that just as easily as prefix search, but for the inverted index the problem then becomes much harder. Somewhat surprisingly, even for this setting, an inverted index, built for an appropriate choice of k-grams as "words", was experimentally shown in [Puglisi 06] to outperform suffix arrays. Furthermore, the application behind our problem definition really calls for prefix search and *not* for infix search. Infix search would return too many, mostly irrelevant matches. For example, when typing "search aut", we are most certainly not looking for completions like "flautist" or "aeronautics". (On the other hand, our algorithm can be easily extended to consider reasonable subwords like the "vector" in "eigenvector"; we can simply add these to the index without increasing the total index size considerably. See Section 5.8.)

Still, as our AutoTree data structure (introduced in the next chapter) shares certain characteristics with suffix arrays, e.g., the need for random accesses, we also compared it experimentally against suffix arrays. The results (favorable for AutoTree) are presented in Section 3.8.

Concerning efficient implementations of search engines (or database systems), there is also lots of work on query optimization via choosing a clever execution plan. For a multi-word query and an inverted index this can, e.g., involve first intersecting the shortest lists to quickly limit the set of candidate matches. However, these approaches do not apply to our fully interactive setting, because there is no choice here but to evaluate the query in a strict order, from left to right.

2.4 Notation

The following notation will be used throughout the dissertation. Although the (few) symbols will usually be explained again in the context where they are used, it is helpful to familiarize oneself with them. They are given here mostly for reference purposes.

N	=	total number of word-in-document pairs (w, d)
m	=	total number of distinct words ("vocabulary")
n	=	total number of documents
L	=	average number of word-in-document pairs in a document, i.e., $L = N/n$
W	=	consecutive words (a "word range") corresponding to a prefix
D	=	matching (sorted) document ids for the previous part of a query
D_w	=	(sorted) document ids for documents containing the word w
Φ	=	matching word-in-document pairs (w, d) for an autocompletion search query
W'	=	$\{w : \exists (w, d) \in \Phi\}$, i.e., the matching completions for an autocompletion search query
D'	=	$\{d : \exists (w, d) \in \Phi\}$, i.e., matching documents for an autocompletion search query

[4] If the number of characters in the collection is N', an uncompressed suffix array needs at least $N' \lceil \log_2(N') \rceil$ bits, which exceeds the $N \lceil \log_2(n) \rceil$ bits required for an inverted index built over the words by a factor of at least the average word length.

Chapter 3

AutoTree Index

In the last chapter, we explained how to use the inverted index to solve autocompletion search queries. In this chapter, we present our first data structure, called AutoTree, for solving such queries. It is designed for use in main memory and makes extensive use of bit vectors. AutoTree has the desirable property that its running time depends, for realistic corpora and queries, linearly on the size of the output. The details are given by the following theorem.

3.1 Main Result

Theorem 1 *Given a collection with n documents, m distinct words, $N \geq 2^5 \cdot m$ word-in-document pairs, and a (constant) average number of distinct words per document $L = N/n$, there is a data structure AutoTree with the following properties:*

(a) AutoTree can be constructed in $O(N)$ time.

(b) AutoTree uses at most $N\lceil \log_2 n \rceil$ bits of space (which is the space used by an ordinary uncompressed inverted index)[1].

(c) AutoTree can process an autocompletion search query (D, W) (according to Definition 1) in time

$$O\left((\alpha + \beta)|D| + \Phi\right),$$

where $\Phi = \sum_{w \in W} |D \cap D_w|$ and D_w is the set of documents containing word w. Here $\alpha = N|W|/(mn)$, which is bounded above by 1, unless the word range is very large (e.g., when completing a single letter), and by L, regardless of assumptions about W. If we assume that the words in a document with l words are a random size-l subset of all words, β is at most 2 in expectation. In our experiments, β is indeed around 2 on the average and about 4 in the (rare) worst case; our analysis implies a general worst-case bound of $\min(\log(mn/N), L_{\max})$, where $L_{\max}$ is the maximum document length.

Note that for constant α and β, the running time is asymptotically optimal, as it takes $\Omega(|D|)$ time to merely read in all of D and it takes $\Omega(\Phi + |W| + |D|) = \Omega(\Phi)$ time to output the result.[2] Also note that asymptotically, as the corpus grows, N, n, m and W will become large but $L_{\max}$, the maximum document length, and hence L, the average document length, can be assumed to remain bounded. In that case, alpha and beta are bounded even in the theoretical worst case. The necessary ingredients for the proof of Theorem 1 are developed in the next sections and they are finally assembled in Section 3.5.

The condition on N is a technicality and is satisfied for any realistic document collection. Details are given in Section 3.5. Intuitively speaking, the condition says that n, the number of documents grows at least as fast as m, the number of terms (assuming that L, the average document length, stays constant). This condition will guarantee that AutoTree requires less space than BASIC, which can be understood intuitively as follows: BASIC only needs to encode, for each word-in-document pair, a single *document* id (neglecting the small

[1] Strictly speaking, an uncompressed inverted index needs even more space, to store the list lengths.

[2] The statement about the required time to read in the (usually "random") set D tacitly assumes D is explicitly represented element-by-element. Of course, for the first prefix, when D is the set of all documents, this is not the case.

overhead for storing the list lengths and the word, a list pertains to). Thus its space requirement directly depends on the number of documents. AutoTree, as we will explain in the following sections, essentially encodes each such pair using its *word* id.

We implemented AutoTree, and in Section 3.6 show that its processing time correlates almost perfectly with the bound from Theorem 1(c) above (for constant α and β). In that Section, we also compare it to the inverted index (see Section 2.2), which AutoTree outperforms by a factor of 10 in worst-case processing time (which is key for an interactive feature), and by a factor of 4 in average-case processing time.

3.1.1 Related Work

Work related to the general autocompletion search problem according to Definition 1, has already been discussed in Section 2.3. Here, we merely discuss data structures with certain similarities to ours, in particular wavelet trees [Grossi 03; Ferragina 06].

A wavelet tree consists of a tree, built over a fixed alphabet, where each node contains a bitvector. These bitvectors are "relative" as the bits in the left/right child of a bit vector in a node correspond to the 1/0 bits in its parent. So the length of a particular bit vector depends on the number of 1/0 bits of its parent node. To allow for constant-time rank and select operations on these bit vectors, auxiliary data structures are built [Munro 96]. Our data structure also makes use of relative bitvectors, but these serve a different purpose than in wavelet trees: in our tree both children of a node store only information corresponding to the 1 bits of their parent node, and *nothing* for 0 bits. Furthermore, an integral part of our data structure is a "witness" stored by each 1 bit (whereas in a wavelet tree one only obtains the final information after descending to the leaf level).

In the description of our data structures we will point out some interesting analogies to the geometric range-search data structures from [Chazelle 88] and [McCreight 85].

3.1.2 Outline of the Rest of This Chapter

In the following sections, we explain the indexing scheme AutoTree, with the properties given in Theorem 1. A combination of four main ideas will lead us to this scheme: a tree over the words (Section 3.2), relative bit vectors (Section 3.3), pushing up the words (Section 3.4), and dividing into blocks (Section 3.5). In Section 3.6, we will complement our theoretical findings with experiments on a large test collection.

3.2 Building a Tree Over the Words (TREE)

The idea behind our first scheme on the way to Theorem 1 is to *increase the amount of preprocessing by precomputing inverted lists not only for words but also for their prefixes*. More precisely, we construct a complete binary tree with m leaves, where m is the number of distinct words in the collection. We assume here and throughout this chapter that m is a power of two. For each node v of the tree, we then precompute the sorted list D_v of documents which contain at least one word from the subtree of that node. The lists of the leaves are then exactly the lists of an ordinary inverted index, and the list of an inner node is exactly the union of the lists of its two children. The list of the root node is exactly the set of all non-empty documents. A simple example is given in Figure 3.1.

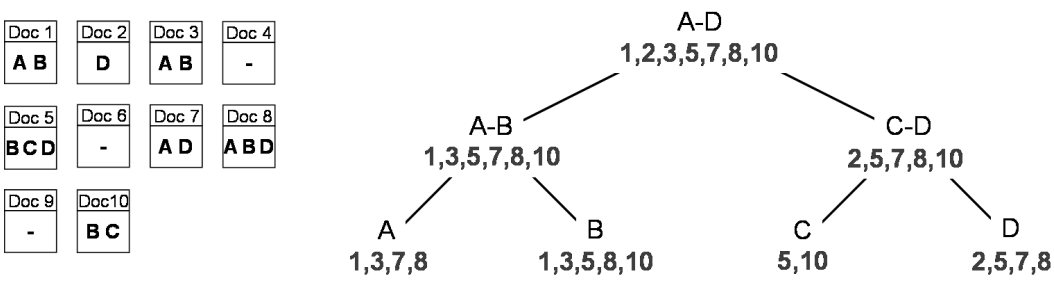

Figure 3.1: Toy example for the data structure of scheme TREE with 10 documents and 4 different words.

Given this tree data structure, an autocompletion search query given by a word range W and a set of documents D is then processed as follows.

1. Compute the unique minimal sequence $v_1, \ldots, v_\ell$ of nodes with the property that their subtrees cover exactly the range of words W. Process these ℓ nodes from left to right, and for each node v invoke the following procedure.

2. Fetch the list D_v of v and compute the intersection $D \cap D_v$. If the intersection is empty, do nothing. If the intersection is non-empty, then if v is a leaf corresponding to word w, report for each $d \in D \cap D_v$ the pair (w, d). If v is not a leaf, invoke this procedure (step 2) recursively for each of the two children of v.

Scheme TREE can potentially save us time: If the intersection computed at an inner node v in step 2 is empty, we know that none of the words in the whole subtree of v is a completion leading to a hit, that is, *with a single intersection we are able to rule out a large number of potential completions*. However, if the intersection at v is non-empty, we know nothing more than that there is *at least one word* in the subtree which will lead to a hit, and we will have to examine both children recursively. The following lemma shows the potential of TREE to make the query processing time depend on the output size instead of on W as for INV. Since TREE is just a step on the way to our final scheme AutoTree, we do not give the exact query processing time here but just the number of nodes visited, because we need exactly this information in the next section.

Lemma 3 *When processing an autocompletion search query (D, W) with TREE, at most $2(|W'| + 1)\log_2 |W|$ nodes are visited, where W' is the set of all words from W that occur in at least one document from D.*

Proof. A node at height h has at most 2^h nodes below it. So each of the nodes $v_1, \ldots, v_l$ has height at most $\lfloor \log_2 |W| \rfloor$. Further, no three nodes from $v_1, \ldots, v_l$ have identical height, which implies that $l \leq 2\lfloor \log |W| \rfloor$. Similarly, for each word in W' we need to visit at most two additional nodes, each at height below $\lfloor \log |W| \rfloor$. ■

The price TREE pays in terms of space is large. In the worst case, each level of the tree would use just as much space as the inverted index stored at the leaf level, which would give a blow-up factor of $\log_2 m$.

3.3 Relative Bitvectors (TREE+BITVEC)

In this section, we describe and analyze TREE+BITVEC, which reduces the space usage from the last section, while maintaining as much as possible of its potential for a query processing time depending on W', the set of matching completions, instead of on W. *The INV trick will be to store the inverted lists via relative bit vectors*. The resulting data structure turns out to have similarities with the static 2-dimensional orthogonal range counting structure of Chazelle [Chazelle 88].

In the root node, the list of all non-empty documents is stored as a bit vector: when N is the number of documents, there are N consecutive bits, and the ith bit corresponds to document number i, and the bit is set to 1 if and only if that document contains at least one word from the subtree of the node. In the case of the root node this means that the ith bit is 1 if and only if document number i contains any word at all.

Now consider any one child v of the root node, and with it store a vector of N' bits, were N' is the number of 1-bits in the parent's bit vector. To make it interesting already at this point in the tree, assume that indeed some documents are empty, so that not all bits of the parent's bit vector are set to one, and $N' < N$. Now the jth bit of v corresponds to the jth 1-bit of its parent, which in turn corresponds to a document number i_j. We then set the jth bit of v to 1 if and only if document number i_j contains a word in the subtree of v.

The same principle is now used for every node v that is not the root. Constructing these bit vectors is relatively straightforward; it is part of the construction given in Section 3.4.1.

Lemma 4 *Let s_{tree} denote the total lengths of the inverted lists of algorithm TREE. The total number of bits used in the bit vectors of algorithm TREE+BITVEC is then at most $2s_{tree}$ plus the number of empty documents (which cost a 0-bit in the root each).*

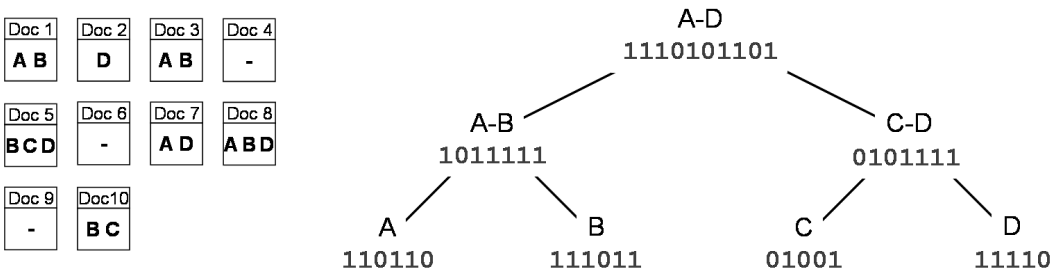

Figure 3.2: The data structure of TREE+BITVEC for the toy collection from Figure 3.1.

Proof. The lemma is a consequence of two simple observations. The first observation is that wherever there was a document number in an inverted list of algorithm TREE there is now a 1-bit in the bit vector of the same node, and this correspondence is 1 − 1. The total number of 1-bits is therefore s_{tree}.

The second observation is that if a node v that is not the root has a bit corresponding to some document number i, then the parent node also has a bit corresponding to that same document, and that bit of the parent is set to 1, since otherwise node v would not have a bit corresponding to that document.

It follows that the nodes, which have a bit corresponding to a particular fixed document, form a subtree that is not necessarily complete but where each inner node has degree 2, and where 0-bits can only occur at a leaf. The total number of 0-bits pertaining to a fixed document is hence at most the total number of 1-bits for that same document plus one. Since for each document we have as many 1-bits at the leaves as there are words in the documents, the same statement holds without the plus one.

■

The procedure for processing a query with TREE+BITVEC is, in principle, the same as for TREE. The only difference comes from the fact that the bit vectors, except that of the root, can only be interpreted relative to their respective parents.

To deal with this, we ensure that whenever we visit a node v, we have the set $\mathcal{I}_v$ of those positions of the bit vector stored at v that correspond to documents from the given set D, as well as the $|\mathcal{I}_v|$ numbers of those documents. For the root node, this is trivial to compute. For any other node v, $\mathcal{I}_v$ can be computed from its parent u: for each $i \in \mathcal{I}_u$, check if the ith bit of u is set to 1, if so compute the number of 1-bits at positions less than or equal to i, and add this number to the set $\mathcal{I}_v$ and store by it the number of the document from D that was stored by i. With this enhancement, we can follow the same steps as before, except that we have to ensure now that whenever we visit a node that is not the root, we have visited its parent before. The lemma below shows that we have to visit an additional number of up to $2 \log_2 m$ nodes because of this.

Lemma 5 *When processing an autocompletion search query (D, W) with TREE+BITVEC, at most $2(|W'| + 1) \log_2 |W| + 2 \log_2 m$ nodes are visited, with W' defined as in Lemma 3.*

Proof. By Lemma 3, at most $2(|W'| + 1) \log_2 |W|$ nodes are visited in the subtrees of the nodes $v_1, \ldots, v_l$ that cover W. It therefore remains to bound the total number of nodes contained in the paths from the root to these nodes $v_1, \ldots, v_l$.

First consider the special case, where W starts with the leftmost leaf, and extends to somewhere in the middle of the tree. Then each of the $v_1, \ldots, v_l$ is a left child of one node of the path from the root to v_l. The total number of nodes contained in the l paths from the root to each of $v_1, \ldots, v_l$ is then at most $d - 1$, where d is the depth of the tree. The same argument goes through for the symmetric case when the *range ends with the rightmost leaf*.

In the general case, where W begins at some intermediate leaf and ends at some other intermediate leaf, there is a node u such that the leftmost leaf of the range is contained in the left subtree of u and the rightmost leaf of the range is contained in the right subtree of u. By the argument from the previous paragraph, the paths from u to those nodes from $v_1, \ldots, v_l$ lying in the left subtree of u then contain at most $d_u - 1$ different nodes, where d_u is the depth of the subtree rooted at u. The same bound holds for the paths from u to the other nodes from $v_1, \ldots, v_l$, lying in the right subtree of u. Adding the length of the path from the root to u, this gives a total number of at most $2d - 3$

■

3.4 Pushing Up the Words (TREE+BITVEC+PUSHUP)

The scheme TREE+BITVEC+PUSHUP presented in this section gets rid of the $\log_2 |W|$ factor in the query processing time from Lemma 5. *The idea is to modify the TREE+BITVEC data structure such that for each element of a non-empty intersection, we find a new word-in-document pair* (w, d) *that is part of the output.* For that we store with each single 1-bit, which indicates that a particular document contains a word from a particular range, one word from that document and that range. We do this in such a way that each word is stored only in one place for each document in which it occurs. When there is only one document, this leads to a data structure that is similar to the priority search tree of McCreight, which was designed to solve the so-called 3-sided dynamic orthogonal range-reporting problem in two dimensions [McCreight 85].

Let us start with the root node. Each 1-bit of the bit vector of the root node corresponds to a non-empty document, and we store by that 1-bit the *lexicographically smallest* word occurring in that document. Actually, we will not store the word but rather its number, where we assume that we have numbered the words from $0, \ldots, m-1$.

More than that, for all nodes at depth i (i.e., i edges away from the root), we omit the leading i bits of its word number, because for a fixed node these are all identical and can be computed from the position of the node in the tree. However, asymptotically this saving is not required for the space bounds in Theorem 1 as dividing the words into blocks will already give a sufficient reduction of the space needed for the word numbers.

Now consider anyone child v of the root node, which has exactly one half H of all words in its subtree. The bit vector of v will still have one bit for each 1-bit of its parent node, but the definition of a 1-bit of v is slightly different now from that for TREE+BITVEC. Consider the jth bit of the bit vector of v, which corresponds to the jth set bit of the root node, which corresponds to some document number i_j. Then this document contains at least one word — otherwise the jth bit in the root node would not have been set — and the number of the lexicographically smallest word contained is stored by that jth bit. Now, if document i_j contains other words, and at least one of these *other* words is contained in H, only then the jth bit of the bit vector of v is set to 1, and we store by that 1-bit *the lexicographically smallest word contained in that document that has not already been stored in one of its ancestors* (here only the root node).
Figure 3.3 explains this data structure by a simple example. The construction of the data structure is relatively straightforward and can be done in time $O(N)$. Details are given in Section 3.4.1.

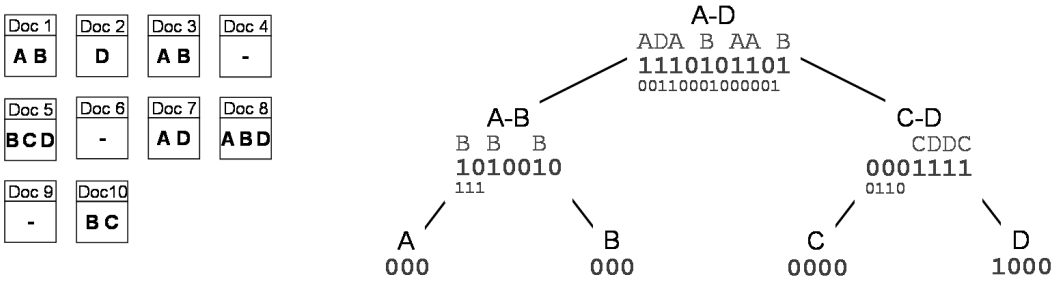

Figure 3.3: The data structure of TREE+BITVEC+PUSHUP for the example collection from Figure 3.1. The large bitvector in each node encodes the inverted list. The words stored by the 1-bits of that vector are shown in gray on top of the vector. The word list actually stored is shown below the vector, where A=00, B=01, C=10, D=11, and for each node the common prefix is removed, e.g., for the node marked C-D, C is encoded by 0 and D is encoded by 1. A total of 49 bits is used, not counting the redundant 000 vectors and bookkeeping information like list lengths etc.

To process a query we start at the root. Then, we visit nodes in such an order that whenever we visit a node v, we have the set $\mathcal{I}_v$ of exactly those positions in the bit vector of v that correspond to elements from D (and for each $i \in \mathcal{I}_v$ we know its corresponding element d_i in D). For each such position with a 1-bit, we now check whether the word w stored by that 1-bit is in W, and if so output (w, d_i). This can be implemented by

random lookups into the bit vector in time $O(|\mathcal{I}_v|)$ as follows. First, it is easy to intersect D with the documents in the root node, because we can simply lookup the document numbers in the bitvector at the root. Consider then a child v of the root. What we want to do is to compute a new set I_v of document indices, which gives the numbering of the document indices of D in terms of the numbering used in v. This amounts to counting the number of 1-bits in the bitvector of v up to a given sequence of indices. Each of these so-called *rank* computations can be performed in constant time with an auxiliary data structure that uses space sublinear in the size of the bitvector [Munro 96].

Consider again the check whether a word w stored by a 1-bit corresponding to a document from D is actually in W. This check can only fail for relatively few nodes, namely those with at least one leaf not from W in their subtree. These checks do not contribute an element to the output set, and are accounted for by the factor β mentioned in Theorem 1, and Lemmas 6 and 8 below.

Lemma 6 *With TREE+BITVEC+PUSHUP, an autocompletion search query (D, W) can be processed in time $O(|D| \cdot \beta + \sum_{w \in W} |D \cap D_w|)$, where β is bounded by $\log_2 m$ as well as by the average number of distinct words in a document from D. For the special case, where W is the range of* all *words, the bound holds with $\beta = 1$.*

Proof. As we noticed above, the query processing time spent in any particular node v can be made linear in the number of bits inspected via the index set $\mathcal{I}_v$. Recall that each $i \in \mathcal{I}_v$ corresponds to some document from D. Then for reasons identical to those that led to the space bound of Lemma 4, for any fixed document $d \in D$, the set of all visited nodes v which have an index in their $\mathcal{I}_v$ corresponding to d form a binary tree, and it can only happen for the leaves of that tree that the index points to a 0-bit, so that the number of these 0-bits is at most the number of 1-bits plus one.

Let again $v_1, \ldots, v_l$ denote the at most $2 \log_2 m$ nodes covering the given word range W (see Section 3.2). Observe that, by the time we reach the first node from $v_1, \ldots, v_l$, the index set $\mathcal{I}_v$ will only contain indices from D', as all the 1-bits for these nodes correspond to a word in W'. Strictly speaking, this is only guaranteed after the intersection with this node, which accounts for an additional D in the total cost. Thus, each distinct word w we find in at least one of the nodes can correspond to at most $|D \cap D_w|$ 1-bits met in intersections with the bitvectors of other nodes in the set, and each 1-bit leads to at most two 0-bits met in intersections. Summing over all $w \in W$ gives the second term in the equation of the lemma.

The remaining nodes that we visit are all ancestors of one of the $v_1, \ldots, v_l$, and we have already shown in the proof of Lemma 5 that their number is at most $2 \log_2 m$. Since the processing time for a node is always bounded by $O(|D|)$, that fraction of the query processing time spent in ancestors of $v_1, \ldots, v_l$ is bounded by $O(|D| \log_2 m)$.

■

Lemma 7 *The bit vectors of TREE+BITVEC+PUSHUP require a total of at most $2N + n$ bits.*

Proof. Just as for TREE+BITVEC, each 1-bit can be associated with the occurrence of a particular word in a particular document, and that correspondence is $1 - 1$. This proves that the total number of 1-bits is exactly N, and since word numbers are stored only by 1-bits and there is indeed one word number stored by each 1-bit, the total number of word numbers stored is also N. By the same argument as in Lemma 4, the number of 0-bits is at most the number of 1-bits plus 1 for each document. This can alternatively be seen as follows: Start with an empty document and, iteratively, insert the lexicographically smallest of its words, which has not been inserted yet. Each such word-in-document pair, which corresponds to a 1 in a bit vector, will be pushed up in the tree as far as possible, thereby replacing one 0-bit and creating (at most) two new 0-bits in its children. Less than two 0-bits (and in fact none at all) will only be created, if the 1-bit was already at the bottom level.

■

3.4.1 The Index Construction for TREE+BITVEC+PUSHUP

The construction of the tree for algorithm TREE+BITVEC+PUSHUP is relatively straightforward and takes *constant amortized time* per word-in-document occurrence (assuming each document contains its word sorted in ascending order).

1. Process the documents in order of ascending document numbers, and for each document d do the following.

2. Process the distinct words in document d in order of ascending word number, and for each word w do the following. Maintain a *current node*, which we initialize as an artificial parent of the root node.

3. If the current node does not contain w in its subtree, then set the current node to its parent, until it does contain w in its subtree. For each node left behind in this process, append a 0-bit to the bit vector of those of its children which have not been visited.

   Note: for a particular word, this operation may take non-constant time, but once we go from a node to its parent in this step, the old node will never be visited again. Since we only visit nodes, by which a word will be stored and such nodes are visited at most three times, this gives constant amortized time for this step.

4. Set the current node to that one child which contains w in its subtree. Store the word w by this node. Add a 1-bit to the bit vector of that node.

3.5 Divide Into Blocks (TREE+BITVEC+PUSHUP+BLOCKS)

This section is our last station on the way to our main result, Theorem 1.

For a given B, with $1 \leq B \leq m$, we divide the set of all words in blocks of equal size B. We then construct the data structure according to TREE+BITVEC+PUSHUP for each block separately. As we only have to consider those blocks, which contain any words from W, this gives a further speedup in query processing time. An autocompletion query given by a word range W and a set of documents D is then processed in the following three steps.

1. Determine the set of ℓ (consecutive) blocks, which contain at least one word from W, and for $i = 1, \ldots, \ell$, compute the subrange W_i of W that falls into block i. Note that $W = W_1 \dot\cup \cdots \dot\cup W_\ell$.

2. For $i = 1, \ldots, \ell$, process the query given by W_i and D according to TREE+ BITVEC+PUSHUP, resulting in a set of matches $M_i := \{(w, d) \in C : w \in W_i, d \in D\}$, where C is the set of of word-in-document pairs.

3. Compute the union of the sets of matching word-in-document pairs $\cup_{i=1}^{\ell} M_i$ (a simple concatenation).

Lemma 8 *With TREE+BITVEC+PUSHUP+BLOCKS and block size B, an autocompletion search query (D, W) can be processed in time $O(|D| \cdot (\alpha + \beta) + \sum_{w \in W} |D \cap D_w|)$, where $\alpha = |W|/B$ and β is bounded by $\log_2 B$ as well as by the average number of distinct words from $W_1 \cup W_\ell$ (the first and the last subrange from above) in a document from D.*

Proof. Let W_i denote the subset of W pertaining to block i. Since each block contains at most B words, according to Lemma 6, we need time at most $O(|D| \log_2 B + \sum_{w \in W_i} |D \cap D_w|)$ for a block i. However, for all but at most two of these blocks (the first and the last) it holds that all words of the blocks are in W, so that according to the special case in Lemma 6, the query processing time for each of the at most $|W|/B$ inner blocks is actually $O(|D| + \sum_{w \in W_i} |D \cap D_w|)$. Summing these up gives us the bound claimed in the lemma. ∎

Lemma 9 *TREE+BITVEC+PUSHUP+BLOCKS with block size B requires at most $2N + n \cdot \lceil m/B \rceil$ bits for its bit vectors and at most $N\lceil \log_2 B \rceil$ bits for the word numbers stored by the 1-bits. For $B \geq mn/N$, this adds up to at most $4N$ for the bit vectors, and $N(4 + \lceil \log_2 B \rceil)$ bits in total. The auxiliary data structure (for the constant-time rank computation) requires at most an additional $N/4$ bits.*

Proof. To count the number of bits in the relative bitvectors, we use the same argument as for Lemma 7: there is exactly one 1-bit for each of the N word-in-document occurrences. The total number of 0-bits is at most the total number of bits in the roots of the blocks (which gives $n \cdot \lceil m/B \rceil$), plus the total number of 1-bits. So the total space for the bit vectors is bounded by $N + n\lceil m/B \rceil + N \leq 2N + n\lceil N/n \rceil \leq 4N$. The space for the word

numbers is exactly $N\lceil \log_2 B \rceil$, as it requires $\lceil \log_2 B \rceil$ bits to encode a word in a block of size B. Finally, as the total length of the bit vectors is bounded by $4N$, we can construct the auxiliary data structure to require at most N bits [Munro 96].[3]

■

With all the required machinery in place, we can now prove Theorem 1. Part (a) of Theorem 1 is established by the construction given in Section 3.4.1. Part (b) of Theorem 1 follows from Lemma 9 by choosing $B = \lceil nm/N \rceil$. This choice of B minimizes the space bound of Lemma 9, and we call the corresponding data structure AutoTree. Note that it is here that we use the fact $N \geq 2^5 \cdot m$, as it ensures $5 + \log_2(nm/N) \leq \log_2 n$ and ultimately $5 + \lceil \log_2 B \rceil \leq \lceil \log_2 n \rceil$. Part (c) of Theorem 1 follows from Lemma 8 and the following remarks. If the words in a document with L words are a random size-L subset of all words, then the average number of words per document that fall into a fixed block is at most 1. In our experiments, the average value for β was 2.2.

As mentioned just before Lemma 6, β counts the number of bitvector lookup operations for a candidate document in D, which do not contribute any element to the result set. If the wordrange W spans multiple tree blocks of size $B = \lceil nm/N \rceil = \lceil m/L \rceil$, then such "useless" bitvector lookups can also occur at the root nodes of the intermediate tree blocks. However, these comparisons are accounted for by the factor α, which bounds the number of such intermediate blocks, and β only counts such bitvector lookups in the boundary blocks, which also contain at least one word not in W. Note that α is trivially bounded by the total number of blocks, which is $\lceil m/B \rceil \leq 1 + L$, which is constant.

Formally, β is defined as the number of bitvector lookups that need to be performed in the boundary blocks (of which there are at most two) for a candidate document in D until either (a) this document can be ruled out as an element of D' (as it contains no valid completions) or (b) a relevant completion is reported from this document (at which point the total number of additional bitvector lookups is bounded by twice the number of matching output elements for this document). A small, constant β thus indicates a strong output-sensitive behavior of the algorithm. Note that β is bounded by $2L_{\max}$, the maximum number of words in any document.

Finally, it remains to explain, how to obtain W' and D' from Φ. Theoretically, this can be done by having two bit vectors of lengths m and n respectively, which are to be reused for all queries. Note that the extra space required is negligible compared to the size of the data structure itself. Then, while inspecting the elements $(w, d) \in \Phi$, we set the bit corresponding to w in the bit vector of dimension m to 1, if it is not set already, and add w to the set W'. In a similar fashion, we proceed for D'. This takes time $\Theta(|\Phi|)$. Finally, to be able to reuse the two bit vectors, we pass through all elements in D' and W' and reset the corresponding bits to 0. These passes take time $O(|\Phi|)$. In the end, the elements of W' and in particular of D' will be unsorted. At first glance, this could cause a problem, as D' will be the input D for following autocompletion queries. But AutoTree does not require the elements of D to be sorted (unlike INV).

In practice, we simply sort the elements $(w, d) \in \Phi$ by w within each block, to obtain the set W'. Similarly, to obtain D', we merge the output lists of elements (w, d) for individual nodes as, if D is sorted, these will be sorted by d.[4] We chose this approach mainly as (i) it makes the use and aggregation of scores easier, (ii) we can, in fact, use the same sorting/scoring methods for documents for INV and AutoTree (which made software maintenance easier), (iii) the absolute time for sorting is small compared to the time to find the matches, and, (iv) the logarithmic factor in the time required for sorting is small compared to constant costs for, e.g., copying and other simple manipulations, so that we still obtain an almost perfect linear correlation with the size of the Φ, even as the size of Φ varies.

3.6 Experiments

We tested both AutoTree and our baseline INV on two corpora. First, on the corpus of the TREC 2004 Robust Track (ROBUST '04), which consists of the documents on TREC disks 4 and 5, minus the Congressional Record [Voorhees 04]. Second, on the English Wikipedia (WIKIPEDIA), using only article pages and not discussion or user pages.

[3] Note that we do not count book keeping information (neither for AutoTree nor for BASIC), such as space needed to store the lengths of the bit vectors (or the lengths of the inverted lists), as this additional space is asymptotically negligible.

[4] Interestingly, the total number of such lists is bounded by $L \cdot 2^{L_{\max}}$, as each of the L blocks contributes at most $2^{L_{\max}}$ non-empty nodes. This is independent of n, m, N or W, which might seem trivial, but which is something that INV fails to achieve.

In both cases, we implemented AutoTree with an optimal block size (according to Section 3.5), which was 4096 for the ROBUST'04 collection and 65,536 for WIKIPEDIA. Block sizes were rounded to the nearest power of two.

The following table gives details on the collections and on the space consumption of the two schemes; as we can see, AutoTree does indeed use no more space (and for both collections, in fact, significantly less) than INV, as guaranteed by Theorem 1.

						bits per word-in-doc pair	
Collection	raw size	n	m	L	B^*	INV	AUTOT
ROBUST'04	1.9 GB	528,025	771,189	219	4,096	20.0	13.9
WIKIPEDIA	6.0 GB	2,363,363	7,138,267	128	65,536	22.0	17.3

Table 3.1: The characteristics of our test collections: n = number of documents, m = number of distinct words, $L = N/n$ = (rounded) average number of distinct words in a document, B^* = space-optimal choice for the block size. The last two columns give the space usage of INV and AUTOT(REE) in bits per word-in-document pair, not including the (small) additional space for storing list lengths and the auxiliary data structure for constant-time rank operations. Both collections satisfy the condition on N for Theorem 1.

For the ROBUST'04 collection, queries are derived from the 200 "old" [5] queries (topics 301-450 and 601-650) of the TREC Robust Track in 2004 [Voorhees 04]. For the WIKIPEDIA collection, we generated 200 queries randomly as follows: For each query we picked a random document with uniform probability and sampled 4 terms of length at least 4 from it. Terms were sampled according to their tf-idf values, i.e., each term had a probability of being sampled proportional to $tf \cdot \log(n/df)$, where tf is the number of occurrences in the given document, n is the total number of documents and df is the number of documents containing this particular term. See Table 3.2 for some examples of such random queries.

Query 1:	highexplosives normal pyrotechnics primarysources
Query 2:	remained growth overview europe
Query 3:	legislatures seats typically apportion
Query 4:	salisbury inheriting westmoreland thomas
Query 5:	italy mayor frazioni baroque

Table 3.2: Five of the random 200 queries generated for the WIKIPEDIA collection. From these queries, we constructed 800 autocompletion search queries as described below.

In both cases, these queries were then "typed" from left to right, taking a minimum word length of 4 for the first query word, and 2 for any query word after the first. From these autocompletion search queries we further omitted those, which would be obtained by simple filtering from a prefix according to the explanation following Definition 1 in the previous chapter. This filtering procedure is identical for AutoTree and INV and takes only a small fraction of the time for the autocompletion search queries processed according to Definition 1, which is why we omitted it from consideration in our experiments. To give an example, for the ad hoc query `world bank criticism`, we considered the autocompletion search queries `worl`, `world ba`, and `world bank cr`. For the ROBUST'04 collection, we considered a total number of 513 such autocompletion search queries. For the WIKIPEDIA collection, exactly 800 such autocompletion queries were obtained (as all of the 200 "raw" queries contained exactly 4 words).

We implemented INV and AutoTree in C++ and measured query processing times on a Dual Opteron machine, with 2 Intel Xeon 3 GHz processors, 8 GB of main memory, running Linux. We measured the time for producing the output according to Definition 1.

As we implemented a sort-based approach to derive the unique elements in D' from Φ (see the remarks at the end of Section 3.5), the time for scoring and ranking would be essentially identical for AutoTree and INV, and would, according to a number of tests, take only a small fraction of the aforementioned processing time. We therefore excluded it from our measurements. For INV, we implemented a fast linear-time intersect, which,

[5] They are "old" as they had been used in previous years for TREC.

in preliminary experiments not reported here, turned out to be faster than its asymptotically optimal relatives [Demaine 00], due to its almost perfect locality of access.

ROBUST'04 (513 queries)							
Scheme	Max	Mean	StdDev	Median	90%-ile	95%-ile	Correl.
INV	14.8 secs	0.22 secs	0.83 secs	0.042 secs	0.39 secs	0.98 secs	0.99
AutoT	1.15 secs	0.07 secs	0.12 secs	0.042 secs	0.17 secs	0.24 secs	0.99
WIKIPEDIA (800 queries)							
Scheme	Max	Mean	StdDev	Median	90%-ile	95%-ile	Correl.
INV	71.9 secs	2.20 secs	7.28 secs	0.351 secs	4.77 secs	10.04 secs	0.99
AutoT	2.17 secs	0.17 secs	0.25 secs	0.032 secs	0.47 secs	0.63 secs	0.99

Table 3.3: Processing times statistics of INV and AutoT(ree) for all queries for both test collections. The 6th and 7th column show the kth worst processing time, where k is 10% and 5%, respectively, of the number of queries. The last column gives the correlation factor between query processing times and total list volume $\sum_{w \in W}(|D| + |D_w|)$ for INV, and input size plus total output volume $|D| + 10 \cdot \sum_{w \in W} |D \cap D_w|$ for AutoTree.

The results from Table 3.3 conform nicely to our theoretical analysis. Four main observations can be made: (i) with respect to maximal query processing time, which is key for an interactive application, AutoTree improves over INV by a factor of more than 10; (ii) in average processing time, which is significant for throughput in a high-load scenario, the improvement is still a factor of 3 for the smaller collection and 13 for the larger collection; (iii) processing times of AutoTree are sharply concentrated around their mean, while for INV they vary widely (in both directions as we checked); (iv) the almost perfect correlation between query processing times and our analytical bounds (explained in the caption of Figure 3.3) demonstrates both the soundness of our theoretical modeling and analysis as well as the accuracy of our implementation.

ROBUST'04 (513 queries)				
	1-word (199 queries)		multi-word (314 queries)	
Scheme	Max	Mean	Max	Mean
INV	0.11secs	0.01secs	14.82secs	0.35secs
AutoTree	0.67secs	0.12secs	1.15secs	0.05secs
WIKIPEDIA (800 queries)				
	1-word (200 queries)		multi-word (600 queries)	
Scheme	Max	Mean	Max	Mean
INV	0.23secs	0.03secs	71.85secs	2.92secs
AutoTree	1.36secs	0.41secs	2.17secs	0.09secs

Table 3.4: Breakdown of query processing for INV and AutoTree by number of query words.

Table 3.4, finally, breaks down query processing times by the number of query words. As we can see, INV is significantly faster than AutoTree for the 1-word queries, however, not because AutoTree is slow, but because INV is extremely fast on these queries. This is so, because INV does not have to compute any intersections for a 1-word query but merely has to copy all relevant lists D_w to the output, whereas AutoTree has to extract, for each output element, bits from its (packed) document id and word id vectors. On multi-word queries, INV has to process a much larger volume than AutoTree, and we see essentially the situation discussed above for the

overall figures.

3.7 Incorporating Positional Information in AutoTree

Our implementation of AutoTree does *not* support the use of positional information. In particular, the use of phrase or proximity search is currently not possible. One obvious approach to remedy this would be to consider each region of interest, e.g., a phrase or, in the extreme case, every position as a separate document and to also keep track of which of these "micro-documents" belong to the same "macro-document", i.e., ordinary document.

However, such an approach would inherently destroy any output-sensitive behavior for the (common) case, where we would like the usual document-level completion, disregarding possible positional information. We have to commit to a certain "unit", for which we want an output-sensitive behavior, and this unit cannot be dynamically changed. A solution involving the duplication of several AutoTrees with different notions of a "document" would lead to an increase of space.

In practice, it might still be feasible to commit to a single unit of granularity and then, for broader queries, merge the results from the "micro-documents". We did, however, not follow this approach to the end, as we started developing our alternative and generally superior (see Section 4.6) HYB data structure, presented in the next chapter. HYB easily allows the incorporation of positional information and also comes with guarantees on the compressibility of this information.

3.8 AutoTree vs. Suffix Arrays

As already discussed in Section 2.3, suffix arrays could also readily be applied to find all occurrences of a given prefix. Here, we will give experimental evidence that suffix arrays do indeed (i) take too much space or (ii) take long to report all occurrences.

We compared AutoTree against a state-of-the art implementation of Succinct Suffix Arrays (version 2) [Mäkinen 04] (SSA2), which can be downloaded from the Pizza&Chili website at `http://pizzachili.dcc.uchile.cl/`. We also experimented with the other available suffix array variants available on that site, but SSA2 performed best and, for the same reason, was chosen as a baseline in [Puglisi 06].

SSA2 comes with two crucial tuning parameters, which govern the space-time trade-off. The sampling rate, which controls the gaps between positions sampled in the full suffix array, and a factor, which governs the amount of space for the auxiliary data structure, which is used for the constant time rank queries for bit vectors.[6] Slightly confusingly, a *low* sampling rate, in the authors' terminology, means that *many* positions are sampled, that the gaps are thus small, and that therefore the space consumption goes up while the processing time goes down. In Table 3.5, we show the performance of SSA2 for various choices of this parameter. We also tried out several values for the factor pertaining to the space of the auxiliary data structure. However, this turned out to have only a very small effect and the default value of 4, so that at most $1/4$-th of the space of a bit vector is used for the constant time rank data structure, tended to give the best performance.

As AutoTree returns documents, and not positions, for a given prefix, we built a single string for our text corpus, where multiple word-in-document occurrence were combined to a single occurrence of that word. This way, the number of occurrences returned by both methods was identical. The corpus, we used for this experiment, contained $n = 51,671$ documents from the domain of homeopathic medicine. In total, there were $N = 13,970,105$ word-in-document pairs and the total size of the (uncompressed) string on disk was 97 MB. We compared the time that both SSA2 and AutoTree take to report all occurrences of 200 random (but occurring) 4-letter prefixes. Note that $|D| = n$ in this case, which constitutes the worst-case for our data structure. As AutoTree cannot easily reproduce the original corpus from its index, whereas SSA2 can, we also counted the size of the string toward the size of AutoTree. Numbers in parentheses in the table refer to the setting, where this string is compressed with gzip, which reduces its size from 97 to 26 MB. The time to sort the occurrences is *not* included in the time for AutoTree, as SSA2 also returns a list of *unsorted* occurrences.

Table 3.5 shows that even in worst case setting for AutoTree, where $|D| = n$ and the full, uncompressed string is counted toward AutoTree's index size, the state-of-the-art SSA2 suffix array needs at least 17% more

[6] AutoTree also uses such an auxiliary data structure.

SSA2 sample rate	(disk space SSA2)/ (disk space AutoTree)	(average time SSA2)/ (average time AutoTree)
64	0.63 (1.15)	14.9
32	0.70 (1.30)	6.89
16	0.86 (1.58)	2.87
8	1.17 (2.17)	0.94
4	1.79 (3.30)	0.21

Table 3.5: Comparison of relative space and time consumption for AutoTree and SSA2. Both data structures were used to find the (unsorted) occurrences of 200 4-letter prefixes, which yields AutoTree's worst case of $|D| = n$. SSA2 was built for all possible sample rates between 64 and 4. The space refers to the space of both data structures on disk. The space for storing the string, representing the corpus, is counted toward space consumption for AutoTree. The numbers in parentheses in the 2nd column refer to the scenario, where this string is compressed using gzip.

space, for an improvement in average running time of a mere 6%. If one counted the space of the compressed string instead, or if one considered cases where $|D| << n$, the comparison would be far more lopsided, which is why we disregarded the use of suffix arrays for our autocompletion search setting.

Chapter 4

HYB Index

4.1 Introduction

In this chapter, we present the data structure, named HYB, which our CompleteSearch engine is built upon. It uses no more space than a state-of-the-art compressed inverted index, and can respond to autocompletion queries as described in Definition 1 within a small fraction of a second, for collections up to about a Terabyte in raw size. HYB is optimized for settings where the index is too large to fit in main memory and resides on disk.

Our main competitor in this chapter is, as before, the inverted index (INV). Other data structures that could be directly applied to our problem either use a lot of space or have other limitations; see the discussion in Section 2.3 and also the comparison to the AutoTree index in Section 4.6. We give a rigorous mathematical analysis of HYB with respect to both space usage and query processing times and we complement the analysis of INV, given in Section 2.2, by analyzing INV's space usage when its document lists are compressed. Our analysis accurately predicts the real behavior on our test collections.

Concerning space usage, we define a notion of *empirical entropy* [Ferragina 05] [Williams 99], which captures the inherent space complexity of an index independent of a particular compression scheme. We prove that the empirical entropy of HYB is essentially equal to that of INV, and we find that the actual space usage of our implementation of the two index data structures is indeed almost equal, for each of our three test collections.

Concerning processing times, we give a precise quantification of the number of operations needed, from which we derive bounds for the worst, best, and average-case behavior of HYB. The corresponding bounds for INV are given in Section 2.2. We also take into account the different latencies of sequential and random access to data [Aggarwal 88].

We compare INV and HYB on three test collections with different characteristics. One of our collections has been (semi-)publicly searchable over the last years, so that we have autocompletion search queries from real users for it. Our largest collection is the TREC Terabyte benchmark with over 25 million documents [Clarke 05].

On all three collections and on all the queries we considered, HYB outperforms INV by a factor of $30-60$ in worst-case query processing time, and by a factor of $5-15$ in average case query processing time. In absolute terms, HYB achieves average query processing of less than one tenth of a second on all collections, on a single machine and with the index on disk (and not in main memory).

4.2 Definition of Empirical Entropy

To analyze the inherent space complexity of INV and HYB independently of the specialties of a particular compression scheme, we introduce a notion of *empirical entropy*, which is supposed to quantify the minimal number of bits required to store the respective data. Both INV and HYB are essentially a collection of (multi)sets and sequences. The following definition gives a natural notion of entropy for each such building block, and for arbitrary combinations of them (similar definitions have been made in [Ferragina 05] and [Williams 99]). The reader might first want to skip the following definition and come back to it when it is first used in the analysis that follows.

Definition 3 *We define* empirical entropy *for the following entities, where* $\mathcal{H}(p_1,\ldots,p_l) = -\sum_{i=1}^{l}(p_i \cdot \log_2 p_i)$ *is the* l*-ary entropy function.*

(a) For a subset *of size* n' *with elements from a universe of size* n*, the empirical entropy is* $n\cdot\mathcal{H}(n'/n, 1-n'/n)$ *(include each element of the universe into the subset with probability* n'/n*), which is*

$$n' \cdot \log_2 \frac{n}{n'} + (n-n') \cdot \log_2 \frac{n}{n-n'}.$$

(b) For a multisubset *of size* n' *with elements from a universe of size* n*, the empirical entropy is* $(n+n')\cdot \mathcal{H}(n'/(n+n'), n/(n+n'))$ *(consider a bitvector of size* $n+n'$*, and let a bit be* 0 *with probability* $n'/(n+n')$ *and* 1 *otherwise; the prefix sums at the* 0*-bits give the multisubset), which is*

$$n' \cdot \log_2 \frac{n+n'}{n'} + n \cdot \log_2 \frac{n+n'}{n}.$$

(c) For a sequence *of* n *elements from a universe of size* l*, where the* i*th element occurs* n_i *times* ($n_1+\cdots+n_l = n$*), the empirical entropy is* $n\cdot\mathcal{H}(n_1/n,\ldots,n_l/n)$ *(for each position, pick element* i *with probability* n_i/n*), which is*

$$n_1 \cdot \log_2 \frac{n}{n_1} + \cdots + n_l \cdot \log_2 \frac{n}{n_l}.$$

(d) For a collection of l *entities with empirical entropies* $\mathcal{H}_1,\ldots,\mathcal{H}_l$*, the empirical entropy is simply* $\mathcal{H}_1 + \cdots + \mathcal{H}_l$*.*

4.3 INV, HYB, and Their Analysis

In this section we will first give a brief recap of INV and then analyze its empirical entropy, as it will serve as a baseline. We then go on and describe HYB, and analyze it with respect to its empirical entropy and its processing time for autocompletion search queries according to Definition 1. Query processing times will be quantified in terms of all relevant parameters; from this we can easily derive worst-case, best-case, and average-case bounds. Our average-case bounds make simplifying assumptions on the distribution of words in the documents, but nevertheless turn out to predict the actual behavior quite well. Section 4.4 compares the empirical entropy of both INV and HYB when (optionally) positional information is available. Implementation issues and the actual performance of our implementations of INV and HYB will be discussed in Section 4.5. We briefly comment on index construction times in Section 4.3.3

4.3.1 Empirical Entropy of INV

The inverted index is described and analyzed in Section 2.2. Simply recall though that, for each word, it stores a sorted list of document (ids) containing this word. In the following, we estimate the inherent space efficiency (empirical entropy) of INV. We do not consider enhancements such as skip pointers [Moffat 96] (which allow for a faster intersection of such lists), which we would expect to give similar benefits for both INV and HYB, however at the price of an increased space usage.

Lemma 10 *Consider an instance of INV with* n *documents and* m *words, and where the* i*th words occurs in* n_i *distinct documents (so that* $n_1 + \cdots + n_m$ *is the total number of word-in-document pairs). Let* $\mathcal{H}_{inv}$ *be the empirical entropy according to Definition 3. Then*

$$\mathcal{H}_{inv} \le \sum_{i=1}^{m}\left(n_i \cdot \frac{1}{\ln 2} + n_i \cdot \log_2 \frac{n}{n_i}\right),$$

and for all collections considered in this paper (where most n_i *are much smaller than* n*) this bound is, in practice, tight up to* 2%*.*

Proof. According to Definition 3 (a) and (d), we have

$$\mathcal{H}_{inv} = \sum_{i=1}^{m} \left(n_i \cdot \log_2 \frac{n}{n_i} + (n - n_i) \cdot \log_2 \frac{n}{n - n_i} \right).$$

To prove the lemma, it suffices to observe that because $1 + x \leq e^x$ for any real x,

$$(n - n_i) \cdot \log_2 \frac{n}{n - n_i} = \frac{n - n_i}{\ln 2} \cdot \ln\left(1 + \frac{n_i}{n - n_i}\right) \leq \frac{n_i}{\ln 2}.$$

Now, for almost all words w_i, we have $n_i << n$. This means that $n_i/(n - n_i)$ is close to zero, which means $1 + x \approx e^x$. For our collections, the (averaged) difference was less than 2%. ■

Lemma 10 tells us that if the documents in each list were picked uniformly at random, then a *Golomb-encoding of the gaps* [Witten 99] from one document id to the next (for list i, the expected size of a gap would be n/n_i) would achieve a space usage very close to $\mathcal{H}_{inv}$ bits. In our implementation, we opted to encode gaps with the Simple-9 encoding from [Anh 05], which is easy to implement, and achieves very fast decompression speeds at the price of only a moderate loss in compression efficacy; details are reported in Section 4.5.

4.3.2 Our New Data Structure (HYB)

The basic idea behind HYB is simple: *precompute inverted lists for prefixes instead of individual words*. Assume an autocompletion search query (D, W), where the union of all lists for word range W have been precomputed. We would then get D' with a single intersection (of D with the precomputed list). However, from this precomputed list alone we can no longer infer the set W' of completions leading to a hit. Since W can be an arbitrary word range, it is also not clear which unions should be precomputed, especially when we do not want to use more space than an (optimally compressed) inverted index.

The analysis given in this section suggests the following approach: group the words in blocks so that the lengths of the inverted lists in each block sum to (approximately) $c \cdot n$, for some constant $c < 1$ (we will later choose $c \approx 0.2$). For each block, store the union of the covered inverted lists as a compressed *multiset*, using an effective gap encoding scheme just as done for INV (repetitions of the same element in the multiset correspond to a gap of zero). In parallel to each multiset, for each element x store the id of the word that led to the inclusion of (this occurrence of) x in the multiset. This gives a sequence of word ids, the length of which is exactly the size of the multiset. Encode these word ids with code length (approximately) $\log_2((n_1 + \cdots + n_l)/n_i)$ for the ith word, where n_i is the number of documents containing the ith word, and l is the number of words in the respective block.

Here is an example. Let one of the blocks comprise four words *A*, *B*, *C*, and *D*, with inverted lists

A:3, 5, 6, 8, 9, 11, 12, 15
B:5, 11
C:3, 7, 11, 13
D:3, 8

We would then like to store, in compressed form, the multiset (of document ids) and the sequence (of word ids)

3 3 3 5 5 6 7 8 8 9 11 11 11 12 13 15
A C D A B A C A D A A B C A C A

The optimal encoding of the words *A*, *B*, *C*, *D* would use code lengths $\log_2(16/8) = 1$, $\log_2(16/2) = 3$, $\log_2(16/4) = 2$, $\log_2(16/2) = 3$, respectively, for example $A = 0$, $B = 110$, $C = 10$, $D = 111$. An optimal encoding of the four gaps 0, 1, 2, 3 that occur in the above multiset of document ids would be 0, 10, 110, 111, respectively. What we actually store are then the two bit vectors (where the | are solely for better readability; the codes in this example are prefix-free)

111|0|0|110|0|10|10|10|0|10|110|0|0|10|10|110
0|10|111|0|110|0|10|0|111|0|0|110|10|0|10|0

Note that due to the two different encodings the two lists end up having different lengths in compressed form, and this is also what will happen in reality.

The following analysis will make very clear that, to obtain bounds on the total space usage, (i) one should choose blocks of equal list volume (and not, for example, of equal number of words), (ii) this volume should be a small but substantial fraction of the number of documents (and neither smaller nor larger), and (iii) the lists of document ids should be "gap-encoded" while the lists of word ids should be "entropy-encoded".

As for the space usage, we will first derive a tight bound on the entropy of HYB, and then show that, somewhat surprisingly, if we only choose the block volume to be a small enough fraction of the number of documents, the entropy of HYB is almost exactly that of INV.

We will then show how HYB can be used to process autocompletion queries in time linear in the number of documents, provided that the blocks are chosen of sufficiently large volume and the given word range is not too large (that is, the prefix to be completed is not too unspecific). In that case, HYB will provide for an excellent locality of access, since the basic operation will be one of scanning long lists.

Lemma 11 *Consider an instance of HYB with n words and m documents, where the ith word occurs in n_i documents, and where for each block the sum of the n_i with i from that block is $c \cdot n$, for some $c > 0$. Then the empirical entropy $\mathcal{H}_{hyb}$, defined according to Definition 3, satisfies*

$$\mathcal{H}_{hyb} \;\leq\; \sum_{i=1}^{m} \left(n_i \cdot \frac{1 + c/2}{\ln 2} + n_i \cdot \log_2 \frac{n}{n_i} \right),$$

and the bound is tight as $c \to 0$.

Proof. Consider a fixed block of HYB, and let n_i denote the number of documents containing the ith word belonging to that block. Throughout this proof, let $\sum_i n_i$ denote the sum over all these n_i (so that the sum over all $\sum_i n_i$ from all blocks gives the $\sum_{i=1}^{m} n_i$ from the lemma). According to Definition 3 (b), (c), and (d), the empirical entropy of this block is then

$$\textstyle\sum_i n_i \cdot \log_2 \dfrac{n + \sum_i n_i}{\sum_i n_i} + n \cdot \log_2 \dfrac{n + \sum_i n_i}{n} + \sum_i n_i \log_2 \dfrac{\sum_i n_i}{n_i}.$$

Now adding the first and the last term, the arguments of the logarithms partially cancel out (!), and we get

$$\textstyle\sum_i n_i \cdot \log_2 \dfrac{n + \sum_i n_i}{n_i} \;+\; n \cdot \log_2 \dfrac{n + \sum_i n_i}{n}.$$

Now using that, by assumption, $\sum_i n_i = c \cdot n$, we obtain

$$\textstyle\sum_i n_i \left((1 + 1/c) \log_2(1 + c) + \log_2 \dfrac{n}{n_i} \right).$$

Since $(1 + 1/c)\ln(1 + c) \leq 1 + c/2$ for all $c > 0$ [1], we can upper bound this (tightly, as $c \to 0$) by

$$\textstyle\sum_i n_i \left(\dfrac{1 + c/2}{\ln 2} + \log_2 \dfrac{n}{n_i} \right).$$

This bounds the empirical entropy of a single block of HYB (the sum goes over all words from that block). Adding this over all blocks gives us the bound claimed in the lemma.

■

Comparing Lemma 11 with Lemma 10, we see that if we let the blocks of HYB be of volume at most $c \cdot n$, for some small fraction c, then the empirical entropy of HYB is essentially that of an inverted index. In Section

[1] To see that $(1 + 1/c)\ln(1 + c) \leq 1 + c/2$ for $c > 0$, first note that the Taylor expansion of $(1 + 1/c)\ln(1 + c)$ about $c = 1$ is $1 + c/2 + \sum_{i=1}^{\infty} c^{2i}(c/((2i + 1) \cdot (2i + 2)) - 1/((2i) \cdot (2i + 1)))$. For $c \leq 1$ all the individual summands are negative, and the total sum is trivially smaller than $1 + c/2$. For $c > 1$, we have $(1 + 1/c)\ln(1 + c) \leq 2\ln(1 + c) \leq 1 + c/2$. This follows from the observations that (i) both $2\ln(1 + c)$ and $1 + c/2$ are strictly increasing in c, (ii) the second derivative of $2\ln(1 + c)$ is always negative, whereas the second derivative of $1 + c/2$ is zero, (iii) the derivative of $2\ln(1 + c)$ is always smaller than that of $1 + c/2$ for $c > 3$, and, finally, (iv) $2\ln(1 + 3) < 1 + 3/2$ for $c = 3$.

4.4, we will see that when we take positional information into account, the empirical entropy of HYB actually becomes *less* than that of INV, for any choice of block volumes.

In our implementation of HYB, we first reduce the sorted lists of document ids to a list of gaps, i.e., differences between doc ids. If we would then encode a gap of size x with a *universal encoding*[MacKay 02] using $\sim \log_2 x$ bits, we would attain the empirical entropy of Definition 2.2. In practice, we encode these gaps using a Simple-9 encoding, just as describe for INV before, which gets very close to the theoretical optimal space consumption while also giving very fast decompression speeds.

For the lists of word ids, entropy-optimal compression could be achieved by techniques such as arithmetic encoding [Witten 99] or Huffman encoding. But, if we assume that the word frequencies in a block have a Zipf-like distribution, i.e. that the i-th most frequent term has a relative frequency proportional to $1/i$, then, as for the gaps, an encoding using $\sim \log_2 x$ for the number x would also be (near) entropy optimal.[2] These frequency ranks we then encode again using the Simple-9 encoding.

Concerning block boundaries, our implementation deviates slightly from the equal-volume strategy suggested by our theoretical analysis. Namely, we group all words with a common prefix of a given fixed length into one block, for example, all words starting with `inf`. This length is fixed such that the average block size is a fraction of the number of documents, as suggested by our analysis. Note that it is not necessary that the block-defining prefixes are all of the same length. We nevertheless choose them this way, accepting a number of very specific prefixes, for example `rts`, with very small blocks. It can be seen from our analysis, that for such small blocks, the gap encoding of the lists of doc ids is suboptimal, but that this effect is attenuated by the fact that word ids from the small words ranges of such blocks can be encoded more efficiently. In practice, we found that choosing blocks based on fixed-length prefixes gives essentially the same index size as if blocks were chosen of a corresponding equal volume.

Lemma 12 *Using HYB with blocks of volume N', autocompletion queries (D, W) can be processed in the following time, where D_w is the inverted list for word w*

$$O\left((|D| + |N'|) \cdot (1 + \lceil \sum_{w \in W} |D_w|/N' \rceil) + \sum_{w \in W} |D \cap D_w| \cdot \log(1 + \lceil \sum_{w \in W} |D_w|/N' \rceil)\right).$$

For $N' = \Theta(n)$ and $|W| \leq m \cdot n/N$, and assuming that the elements of D, D_w, and W are picked uniformly at random from the set of all n documents or all m words, respectively, the expected *processing time is bounded by $O(n)$.*

Proof. According to Definition 1, we have to compute, given (D, W), the set W' of words from W contained in documents from D, as well as the set D' of documents containing at least one such word. For each block B containing at least one word from W, a straightforward linear intersection of the given D with the list of document-word pairs from B, gives us in time $O(|D| + |N'|)$ the set W'_B of all words from W' from block B, as well as the set D'_B of all documents from D' which contain a word from B.

From these, D' can be computed by a k-way merge in time $O(\sum_{w \in W} |D \cap D_w| \log(k))$, where k is the number of blocks that contain a word from W, and W' can be computed by a simple linear-time sort into W buckets (because W is a range). Combining the time for intersection and merging, the total time is $O(k \cdot (|D| + |N'|) + \sum_{w \in W} |D \cap D_w| \log(k))$.

The number k of blocks is at most $1 + \lceil \sum_{w \in W} |D_w|/N' \rceil$, which gives the general statement of the lemma. The randomness assumptions stated in the lemma simply mean that the proportion of word-in-doc pairs from words in W is proportional to W. In other words, doubling $|W|$ results in (roughly) doubling $\sum_{w \in W} |D_w|$. Given this assumption $\sum_{w \in W} |D_w| = O(|W|N/m)$. If the prefix corresponding to the word range W is not too unspecific (such as "a*" or "t*"), then $|W| \leq m \cdot n/N$ and we have $O(|W|N/m) = O(n)$. Thus, for $N' = \Theta(n)$ the expected number k of blocks is constant and the total processing time is $O(n)$. ■

4.3.3 Index Construction Time

Getting from a collection of documents (files) to INV is essentially a matter of one big external sort [Witten 99]. This can, e.g., be done in batches, where in one round we read as much data as fits into main memory, and then

[2] This can be easily seen using the definition of the entropy and the two approximations (i) $\sum_{i=1}^{k} 1/i \approx \ln(k)$ and (ii) $\sum_{i=1}^{k} \log_2(i)/i \approx 0.5 \ln^2(k)/\ln(2)$

we sort and compress this, and we write it back to disk. At the end, we merge the individual lists [Heinz 03]. HYB does not require a full inversion of the data, but can be implemented with only two passes over the data: one pass for a rough sorting by block index, and another pass for sorting the blocks. For our experiments, however, we built the compressed indices for both INV and HYB from an intermediate fully inverted text version of the collection, which takes essentially the same time for both.

4.4 Empirical Entropy with Positional Information

If positional information of each word occurrence is available, then, for the same word, different occurrences in the same document can be distinguished, and phrase search and proximity search, where the search terms are required to appear close to each other, becomes possible. This feature is discussed in more detail in Section 5.5. Here, we are mostly concerned with the additional space requirements.

It is not hard to extend both INV and HYB to accommodate positional information: in the document lists (INV and HYB) as well as in the word lists (HYB only), we duplicate each entry as many times as it occurs in the corresponding document, and store the positions in a parallel array of the same size. Word and document lists are compressed just as before, and the lists of positions are gap-encoded by Simple-9, just like the lists of document ids. The intersection routine is adapted to consider a proximity window as an additional parameter.

We can extend our analysis from Section 4.3 to give bounds on the relative space consumption of INV and HYB. As we will see in Section 4.5.3, the position lists increase the index size by a factor of 4-5, for both INV and HYB (without any kind of stopword removal).

Lemma 13 *Let N_i be the total number of occurrences of the ith words, and let N be the total number of word occurrences. Then*

$$\mathcal{H}_{hyb^*} \;\le\; \mathcal{H}_{inv^*} \;\le\; \sum_{i=1}^{m} \left(N_i/\ln 2 + N_i \cdot \log_2(N/N_i)\right),$$

where $\mathcal{H}_{inv^*}$ and $\mathcal{H}_{hyb^*}$ denote the empirical entropy of INV and HYB, respectively, with positional information. That is, with positional information, HYB is always more space-efficient than INV, irrespectively of how we divide into blocks.

Proof. According to Definition 3, the empirical entropy of INV *with positional information* is

$$\begin{aligned}\mathcal{H}_{inv^*} &= \sum_{i=1}^{m} N_i \cdot \log_2 \frac{N}{N_i} + \sum_{i=1}^{m} (N - N_i) \cdot \log_2 \frac{N}{N - N_i} \\ &\le \sum_{i=1}^{m} \left(N_i/\ln 2 + N_i \cdot \log_2(N/N_i)\right).\end{aligned}$$

The last inequality follows from $\ln(1+x) \le x$. If we incorporate positional information into HYB, the empirical entropy for a single block with volume $\sum_i N_i$ becomes

$$\begin{aligned}\mathcal{H}_{hyb^*} &= \sum_i N_i \cdot \log_2 \frac{N}{\sum_j N_j} + (N - \sum_i N_i) \cdot \log_2 \frac{N}{N - \sum_j N_j} \\ &+ \sum_i N_i \cdot \log_2 \frac{\sum_j N_j}{N_i} \\ &= \sum_i N_i \cdot \log_2 \frac{N}{N_i} + (N - \sum_i N_i) \cdot \log_2 \frac{N}{N - \sum_j N_j},\end{aligned}$$

where the summand $\sum_i N_i \cdot \log_2(\sum_j N_j/N_i)$ is the empirical entropy of the word ids. Summing the empirical entropy over all blocks, using $\sum \sum_i N_i = N$ and the fact that the function $f(x) = (N - x) \cdot \log_2 \frac{N}{N-x}$ is concave, it is easy to see that

$$\sum_{i=1}^{m} (N - N_i) \cdot \log_2 \frac{N}{N - N_i} \ge \sum (N - \sum_i N_i) \cdot \log_2 \frac{N}{N - \sum_j N_j},$$

and hence $\mathcal{H}_{hyb^*} \leq \mathcal{H}_{inv^*}$. ■

In practice, $N_i \cdot \log_2(N/N_i) \approx 2N_i \cdot \log_2(n/n_i)$ and since on average a word occurs about 2-3 times in a document, this is just 4-5 times $n_i \log_2(n/n_i)$, which was the corresponding term in the entropy bound for INV or HYB without positional information (Lemmas 10 and 11).

4.5 Experiments

We implemented both INV and HYB in compressed format, as described in Sections 2.2 and 4.3.2. Each index is stored in a single file with the individual lists concatenated and an array of list offsets at the end. The vocabulary (which is the same for INV as for HYB) is stored in a separate file. All our code is in C++. All our experiments were run on a Dual Opteron machine, with 2 Intel Xeon 3 GHz processors, 8 GB of main memory, and running Linux. We ensured that the index was not cached in main memory.

4.5.1 Test Collections

We compared the performance of INV and HYB on three collections of different characteristics. The first collection is a mailing-list archive plus several encyclopedias on homeopathic medicine (`http://www.homeonet.org`). This collection has been searchable via our engine over the past two years by an audience of several hundred people. The second collection consists of a complete dump of the English Wikipedia from February 2007 (`http://search.mpi-inf.mpg.de/wikipedia_en`). The third collection is the large TREC Terabyte collection [Clarke 05], which served as a stress test for our index structures (and for the authors as well). Details about all three collection are given in Table 4.1, where the "raw size" of a collection is the total size of the original, uncompressed files in their original formats (e.g., HTML or PDF).

4.5.2 Queries

For the Homeopathy collection, we used 47,509 queries from a fixed time slice of our query log for that collection. In most cases, each such query was part of a sequence such as `aci`, `acid`, `acidu`, `acidum`, `acidum pho`, and `acidum phos`, but in some cases, when the user typed very fast or just pasted the whole query, a query might just be an individual full word. For the Wikipedia collection, autocompletion queries were generated from a set of 100 randomly generated queries. For each query we picked a random document with uniform probability and sampled between 1 and 5 terms of length at least 4 from it. Terms were sampled according to their tf-idf values, i.e., each term had a probability of being sampled proportional to $tf \cdot \log(n/df)$, where tf is the number of occurrences in the given document, n is the total number of documents and df is the number of documents containing this particular term. Furthermore, the terms were required to lie within a small enough window as to ensure at least one proximity hit. The number of query terms for these queries was chosen with a mean of 2.2 and a median of 2, which are realistic values for web search queries [Spink 02]. These raw queries where then "typed" from left to right, using a minimal prefix length of 4 for the first term and 3 for later terms. So the raw query "slang alcohol" would yield the autocompletion queries `slan`, `slang`, `slang alc`, `slang alco` etc. For the Terabyte collection, autocompletion queries were generated in again the same way but with a minimal prefix length of 4 for each query word, from the (stemmed) 50 ad-hoc queries of the Robust Track Benchmark [Clarke 05], e.g., `squirrel control protect`. For all collections, where we did not generate queries artificially, we removed queries containing words that had no completion at all in the respective collection, as such queries would trivially lead to empty result sets. For the generated queries, such words were not contained in the first place.

For Homeopathy we included full positional information (according to Section 4.4) in the index and some of the queries were indeed proximity queries. (But even for the others queries positional information was used for the ranking.) For Wikipedia, *all* queries were run as proximity queries (using again a full positional index), while for Terabyte, they were executed as ordinary document-level queries. For all collections, both completions and hits were ranked as we described it in Section 5.4.

Each autocompletion query was processed according to Definition 1, e.g., for `acidum pho`, we compute all completions of `pho` that occur in a document which also contains a word starting with `acidum`, as well as the set of all such documents. The result for each autocompletion query is remembered in a *history*, so that we do not

Collection	Homeopathy	Wikipedia	Terabyte
Raw size	**599 MB**	**9.0 GB**	**426 GB**
#documents	51,670	2,698,964	25,204,103
#words	287,283	7,762,159	25,263,176
#items	14 [31] million	0.3 [0.8] billion	3.5 billion
Vocabulary	3.2 MB	122 MB	239 MB
Entropy	6.6 [13.1] bits	8.8 [14.0] bits	8.4 bits
INV			
index size	**16 [81] MB**	**0.7 [2.4] GB**	**4.6 GB**
-per item	9.1 [21.5] bits	15.6 [25.5] bits	11.0 bits
HYB			
index size	**18 [70] MB**	**0.7 [2.1] GB**	**4.9 GB**
-per item	10.9 [18.9] bits	15.1 [22.9] bits	11.6 bits
-per doc	2.3 [12.4] bits	7.8 [17.9] bits	7.6 bits
-per word	8.6 [6.5] bits	7.3 [5.0] bits	4.0 bits

Table 4.1: Properties of our three test collections, and the space consumption of INV versus HYB. The entries in square brackets are for a full positional index, without any word whatsoever removed.

need to recompute the set of documents matching the first part of the query. E.g., when processing `acidum pho`, we can take the set of documents matching `acidum` from the history; see the explanation following Definition 1.

We nevertheless include the filtered queries in our experiments here, because in reality we will always get a mix of both kinds of queries. Table 4.3 will provide figures for just the difficult (unfiltered) queries. We remark that the history is useful also for caching purposes, but in our experiments we used it solely for the purpose of filtering. See Section 11.8 for other uses of the history.

4.5.3 Index Space

Table 4.1 shows that INV and HYB use essentially the same space on all three test collections. For a full positional index, HYB is slighter more compact than INV, while for an index without positional information it is the other way round. This is exactly what Lemmas 10 and 11, and the derivation in Section 4.4 predicted! The sizes for both INV and HYB exceed that predicted by the empirical entropy by about 50%. This is due to our use of the Simple-9 compression scheme, which trades very fast decompression time for about this increase in space usage [Anh 05]. A combination of Golomb and arithmetic encoding would give us a space usage closer to the empirical entropy. However, decompression would then become the computational bottleneck for almost all queries, see Table 4.3. We remark that, by the way we did our analysis, any new compression scheme with improved compression ratio/decompression speed profile, would immediately yield a corresponding improvement for both INV and HYB.

4.5.4 Query Processing Time

Table 4.2 shows that in terms of query processing time, HYB outperforms INV by a large margin on all collections. With respect to maximum processing time, which is especially critical for an interactive application, the improvement is by a factor of 30-50. With respect to average processing time, which is critical for throughput in a high-load scenario, the improvement is by a factor of 5-10. Note that for all collections the majority of

Collection	Method	mean	90%	99%	max
Homeopathy	INV	0.029 secs	0.012 secs	0.477 secs	24.92 secs
	HYB	0.005 secs	0.016 secs	0.062 secs	0.540 secs
Wikipedia	INV	0.404 secs	0.207 secs	11.19 secs	71.11 secs
	HYB	0.031 secs	0.098 secs	0.423 secs	0.975 secs
Terabyte	INV	1.356 secs	0.933 secs	46.74 secs	72.32 secs
	HYB	0.077 secs	0.232 secs	1.290 secs	2.105 secs

Table 4.2: Average, 90%-ile, 99%-ile and maximum processing times in seconds for INV versus HYB on our three test collections. The statistics are for both filtered and unfiltered queries (see comments after the Definition Section 1).

queries is answered by filtering a previous result (see the comments after Definition 1). Since this filtering is identical for both INV and HYB, the median processing times are identical and are not given in Table 4.2.

The reason that for the Homeopathy collection HYB's 90%-ile processing time is actually *larger* than INV's 90%-ile time, stems from the fact that for this collection the queries (taken from a real log file) also contain several instances of long query words with 5 or more letters, which are are not of a sequence of queries (such as `sy`, `sym`, `symp`, ..., `sympton`) and could thus not be computed by filtering. For these long prefixes with a very small word range W, INV has to process a significantly smaller data volume than HYB and is thus faster. However, HYB's worst case behavior is clearly superior, so that this phenomenon disappears for the mean, the 99%-ile and the maximum processing time, where INV suffers from its very poor performance on some hard queries.

Table 4.3 gives interesting insights into where exactly INV loses against HYB. The table shows a breakdown of the running times of those queries for the Terabyte collection, which were not answered by filtering as discussed above. (Note that the breakdown of the filtered queries would be identical for both methods.) The table differentiates between *one-word queries* like `squi`, `squir`, etc. and *multi-word queries* like `squirrel contr` or `squirrel control prot`.

For the one-word queries, no intersections have to be computed for either INV or HYB. The relevant lists merely have to be copied into a buffer and then be sorted (for INV) or have their words checked for containment in the relevant prefix-range (for HYB). According to Lemma 2, the merging of the intersections then dominates for INV, and this indeed shows in the first column of Table 4.3. For multi-word queries, the result volume $\sum_w |D \cap D_w|$ (Lemmas 2 and 12) goes down, and, according to Lemma 2, the intersection costs dominate for INV, which shows in the third column of Table 4.3. In contrast, columns two and four demonstrate that HYB achieves a better balance of the costs for reading, uncompressing, and intersecting, and none of these essential operations becomes the bottleneck. HYB avoids merging altogether since, by construction, the potential completions from the given word range W always lie within a single block.

The read time of HYB is about 35% larger than that of INV. This is partly because HYB's total volume is slightly larger, so that for each query more data is read, and partly because INV's data is read in (consecutive) chunks, so that the disk cache can perform some precaching, i.e., while the current lists is used in main memory the hard disk can, in parallel, already prefetch bits of the next list and bring it into the disk cache, thus interleaving I/O and computation times. The reason why HYB spends more time decompressing than INV has two reasons: one is that HYB has to handle (slightly) more data; the other is that decompression of the word ids takes 40% longer (for the same compressed input volume) than decompression of the document ids. As we remarked in Section 5.4, the absolute time for ranking is the same for both methods. Ranking takes more time on average for the one-word queries, because these tend to have larger result sets; for HYB this time dominates everything else. Note that the time for ranking includes the time for duplicate removal (as we want to only present matching completions/hits once to the user). The comparison with the time needed for the maintenance of the history, which is nothing but memory allocation and copying, shows that all of HYB's operation are essentially fast list scans.

Query size	1-word		multi-word	
Index type	INV	HYB	INV	HYB
average time	0.45 secs	0.18 secs	6.08 secs	0.17 secs
read	.02 secs 4%	.03 secs 15%	.04 secs .7%	.06 secs 32%
decompress	.01 secs 2%	.02 secs 8%	.02 secs .3%	.04 secs 21%
intersect	——	——	5.94 secs 98%	.04 secs 25%
merging	.26 secs 59%	——	.02 secs .3%	——
ranking	.11 secs 24%	.11 secs 62%	.01 secs .1%	.01 secs 4%
history	.03 secs 6%	.03 secs 15%	.02 secs .3%	.03 secs 15%

Table 4.3: Breakdown of average processing times for INV and HYB, for the difficult (unfiltered) queries on Terabyte.

4.6 AutoTree vs. HYB - Experimental Comparison

Given that we have introduced two new data structures, AutoTree (see the previous Chapter 3) and HYB, for the autocompletion problem, the natural question arises, which one is better. Both data structures were designed with very different goals in mind. AutoTree was designed for main memory, as it heavily depends on random accesses (bit vector lookups) and its goal was to have theoretically appealing properties with respect to its running time, namely an output-sensitive behavior. HYB was designed for external memory and is based on linear scans. It has a non-negligible minimum processing time even for empty queries, but reduces the worst case running time.

In our experiments we examined the differences along two dimensions. First, we ran experiments both for internal and external memory[3]. Second, we used two slightly different query sets. The first query set corresponded to a large $|W|$ but small $|D|$, a setting where AutoTree is expected to shine. The second query set had a less skewed distribution with respect to $|W|$ and $|D|$. The underlying collection was, again, the ROBUST collection (see Section 3.6 for details). Queries were derived in the same manner as described in that section but, to obtain the two different sets, we used different prefix lengths. For the first query set, which is identical to the one in Section 3.6, we used a prefix length of 4 for the first prefix and 2 for the later ones. The (long) prefix length of 4 for the first term, filters out some very short 3-letter words and ensures that, when the full word is typed, the set D' (and hence the new D) will not be too large. The (short) prefix length of 2 for the latter query words, gives a large $|W|$ for these autocompletion queries. This query set is labeled "4-2" in Table 4.4. The second query set, labeled "3-3", uses a prefix length of 3 for all prefixes. Table 4.4 shows the factor in mean processing times for both AutoTree as compared to HYB, for both query sets, and for both external and internal memory. Note that for the first query set a HYB index with blocks chosen for prefix length 2 was built, whereas for the second query set an index for prefix size 3 was built.

The main observation from Table 4.4 is that HYB is, in terms of average processing time, never worse than AutoTree. But it can also be seen that, somewhat surprisingly, AutoTree does hardly suffer, in relative performance compared to HYB, when the index resides on disk. The reason for this is that, on the average, the time for reading the whole data of a single block of AutoTree from disk (and each query required data from one or at most two blocks) is dominated by the computation time of a query, most notably the time for the random bit vector lookups (see Section 3.4). That is, on the average, query processing with AutoTree is not IO-bound, at least not for the collection sizes we have experimented with. Since the number of operations required for a query is proportional to its input + output size, and that in turn is on average roughly proportional to the size of a block of AutoTree, we would expect the same behavior for larger collections too. The observations just made hold true on the average only. The smaller the output size of a query, the more IO-bound the processing is going

[3]To ensure that the data was not cached by the operating system, and really had to be read from disk, before each experiment we read two different very large (20 GB) files from disk several times in a row. Within each experiment (running all queries for a collection), nothing was done to prevent caching by the operating system though.

query type	internal/external memory	(average time AutoTree)/ (average time HYB)
4-2	internal	1.2
	external	1.5
3-3	internal	3.5
	external	4.0

Table 4.4: A relative performance comparison between AutoTree and HYB for the ROBUST'04 collection and for two types of queries. HYB's average running time is always better than AutoTree's.

to be. In a worst case, each step of the algorithm (inspection of a word-in-document pair) might incur one disk seek. In absolute times, both HYB and AutoTree are about 15% slower, when the index resides on disk. This goes up to about 40%, when the time for scoring, which is always done in main memory, is not included in the processing time.

It should also be mentioned that it is easy to find individual queries where both AutoTree and HYB outperform their "opponent" by a factor of $4 - 10$. E.g., AutoTree will be preferable for 2-word queries, where the first part is very specific (`environmentalist`) and the second part is extremely broad (`co`). Still, overall, HYB turned out to be the more performant data structure. This, along with the fact that the integration of positional information is easier, was the reason we built our CompleteSearch engine on top of HYB and the experiments in the following chapters are done for HYB only.

Chapter 5

Autocompletion Search and Simple Extensions

5.1 Introduction

In Chapter 1 we already gave a few hints, why the basic autocompletion feature might be useful in practice. Here, in the following Section 5.2, we discuss the virtues of this feature a bit more. In Section 5.3, we look at search engines and other systems offering a related feature. Then we go on to discuss various extensions, such as ranking (Section 5.4), proximity search (Section 5.5), OR and NOT operators (Section 5.7) and completions for subwords or to phrase (Section 5.8). These extensions are partly essential to a good search engine (such as ranking), but are not related to the basic autocompletion search feature or the problem given in Definition 1. Features discussed in the following chapters, starting with faceted search in Chapter 6, are of a different nature. They require the core functionality of the CompleteSearch engine for an efficient realization.

5.2 Autocompletion Search Revisited

The basic autocompletion search feature was already introduced in Chapter 1, where also some of its possible advantages were discussed. The algorithmic core was abstracted in the Definition 1. Here we more closely examine the possible benefits of this feature before discussing related systems in Section 5.3.

Recall that autocompletion search incorporates two theoretically unrelated features. First, prefix search (with an instant display of the results) and, second, a display of relevant completions (for the last prefix).

As an example, for a case where this feature might be useful, suppose a user is searching the Wikipedia for information about the current pope. However, either he has forgotten the exact name or he is unsure about spelling variants (“benedict”, “benedikt”, “benedictus”, ...). He starts typing the query `pope bene` and is then presented with a list of matching documents, as well as a list of matching completions, which occur in documents also containing the term “pope”. He now has the option of (a) refining his search further by continuing to type his query or by selecting a particular spelling variant, or (b) going through the result list for the broader query as it is. Figure 1.1 (in Chapter 1) shows a user interface which provides this feature along with a number of extensions.

This feature is useful in a variety of ways. It saves typing. It spares the user the experience of overspecifying the query, when already a (much) shorter query would give the desired result. It helps the user exploring formulations used in the collection, substantially reducing the amount of guess work required. Note that without the context-sensitivity this feature would lose most of its worth; for example, there are hundreds of completions of `bene`, but only few that make sense in the context of words starting with `pope`.

5.3 Related Work

One of the early uses of an automatic prefix completion mechanism was in the Unix Shell, where pressing the tabulator key gives a list of all file names that start with whatever has been typed on the command line after the last space. Nowadays, we find a similar feature in most text editors, and in a large variety of browsing GUIs.

Even when it comes to web search, similar features exist. E.g., Google Suggest (`http://www.google.com/webhp?complete=1`) offers completions coming from a precompiled list of popular queries.[1] This way a "typical" user only has to type `br` to be presented with the completion `britney spears`. The two main differences to our feature are that (i) Google's completions do not come from the full text of the documents (e.g., the phrase `autocompletion search`, which occurs in web documents, is not be suggested, even when typing `autocompletion sear`) and (ii) no results are presented until a particular completion is chosen. Still, in an attempt to save typing effort for the typical user, this approach is certainly sensible. Observe that for these kinds of applications we can easily achieve fast response times by two binary or B-tree searches in the (pre)sorted list of candidate strings.

AlltheWeb Livesearch (`http://livesearch.alltheweb.com/`) offers the very same feature, but additionally automatically launches a web search for the most prominent completion. This way, e.g., a user only has to type `br` to launch a web search for `britney spears`. Neither Google Suggest nor AlltheWeb Livesearch perform a prefix search over the full document collection.

Copernic Desktop Search (`http://www.copernic.com/en/products/desktop-search/`) offers such a prefix search feature for the collection of private documents kept on the local hard drive of a PC. However, even for a collection of roughly 30,000 text documents, the search takes about half a second (which is not slow but not comparable to CompleteSearch), no list of completions is provided and it is not possible to rank the results by relevance score, but only by, e.g., date or size.

The autocompletion feature as described so far is also reminiscent of *stemming*, in the sense that by stemming, too, prefixes instead of full words are considered [Witten 99]. But unlike stemming, our autocompletion feature gives the user feedback on which completions of the prefix typed so far would lead to highly ranked documents. The user can then assess the relevance of these completions to his or her search desire, and decide to (i) type more letters for the last query word (for example, continue typing the query from Figure 1.1 as `information retriev`); or (ii) to start a new query word (for example, continue typing `information ret data`); or (iii) click on one of the hits displayed on the right side, in case it looks promising. There is no way to provide this feature by a stemming preprocessing This kind of user interaction is well known to improve retrieval effectiveness in a variety of situations [Voorhees 94].

While our autocompletion feature is for the purpose of *finding information*, autocompletion has also been employed for the purpose of *predicting user input*, for example, for typing messages with a mobile phone, for users with disabilities concerning typing, or for the composition of standard letters [Bickel 05] [Grabski 04] [Stocky 04] [Darragh 90] [Jakobsson 86]. In [Finkelstein 01], contextual information has been used to select promising extensions for a query. Paynter et al. have devised an interface with a zooming-in property on the word level, based on the identification of frequent phrases [Paynter 00]. We get a related feature by the subword/phrase-completion mechanism described in Section 5.8.

5.4 Ranking

So far, we have considered the following problem (from Definition 1): while the user is typing a query, compute after each keystroke the list of *all* completions of the last query word that lead to at least one hit, as well as the list of *all* hits that would be obtained by any of these completions. In practice, only a *selection* of items from these lists can and will be presented to the user, and it is, of course, crucial that the most relevant completions and hits are selected.

The standard approach for this task in ranked keyword search is as follows [Witten 99]. Have a precomputed *score* for each word-in-document pair. For a given query, compute for each document an aggregation of the scores pertaining to the occurrences of the query words in that document. Return the documents with the highest such aggregated score to the user. All of INV, AutoTree and HYB can be easily adapted to implement any such scoring and aggregation scheme: store by each word-in-document pair its precomputed score, and when intersecting, aggregate the scores.

In our setting, however, we have to deal with two issues which are not present in standard ranked retrieval. The first is that we do not only have to rank matching *documents* from D' but also matching *words* from W'. The second issue is that the score aggregation is now a two-step process. Consider the document scores. In ordinary ranked retrieval, we obtain them by aggregating scores from the individual query terms. In our setting,

[1] We remark that a prototype of our engine already existed when Google Suggest was launched.

each such query term in fact corresponds to a whole range of words, the scores of which we have to reconcile first. The impact of this dual scoring mechanism on search result quality is by itself an interesting topic of research, but beyond the scope of this dissertation.

Our implementation allows for a plug-in of arbitrary functions for the various aggregations. Our default aggregation to reconcile the scores of different completions of the same prefix is as follows: when merging the intersections (which gives the set D' according to Definition 1), compute for each document in D' the *maximal* score achieved for some completion in W' contained in that document, and compute for each completion in W' the sum of the *maximal* scores achieved for this completion for each of the elements from D'.

Our default aggregation of the scores from the various parts (prefixes) of the query to give the final score of a document is essentially by summing the individual contributions. But we (i) promote matches of words in the title; (ii) promote matches of two words close to each other, see Section 5.5; and (iii) promote exact word matches (as opposed to strict prefix matches). These heuristics work well on our test collections, since they are *spam-free* in the sense that title information or the number of matching occurrences of a word are reliable indicators of relevance.

Concerning space consumption, we use a single byte to hold individual scores of a word-in-document (or word-at-position) pair. The more significant bits are used to mark title matches or proximity matches.

For the basic Definition 1 and our theoretical analysis, both concerning (i) running time and (ii) space consumption, we factored out the issue of ranking. This is because, (i) asymptotically the inclusion of ranking does not affect the time bounds derived in Lemmas 2 and 12, and our experiments show that ranking rarely takes more than half of the total query processing time and (ii) the required extra space is identical for INV, AutoTree and HYB. However, even without this extra information one can do basic scoring by counting occurrences and aggregating these counts in various ways as discussed above.

5.5 Proximity/Phrase Search

With a properly chosen scoring function, such as the one outlined above, mere ranking by score aggregation often gives very satisfactory precision/recall behavior. There are many queries, however, where the decisive cue on whether a particular document is relevant or not lies in the fact whether certain of the query words occur *close to each other* in that document. See [Metzler 04] for a recent positive result on the use of proximity information in ad-hoc retrieval.

The proximity operator increases the of our autocompletion feature, because the use of this operator will strongly narrow down the list of completions displayed to the user, which in turn makes it easier for the user to filter out irrelevant completions. For example, when searching the Wikipedia collection the most relevant completion for the non-proximity query `max pl` would be `place` (because `max` and `place` are both frequent words), but for the proximity query `max..pl` it is `planck`. Here the two dots `..` indicate that words should occur within x words of each other, for some user-definable parameter x. In fact, we chose our score aggregation function such that phrase or proximity matches get a higher score, even if the user did not explicitly specify a proximity requirement for a match.

Recall that, when positional information is included in the index, HYB requires less space than INV. See Section 4.4 for a proof of this. AutoTree, on the other hand, has certain inherent problems with using positions information, at least if the desirable output-sensitivity should to be preserved. See Section 3.7 for details.

5.6 Structured Search in XML Documents

Any kind of full-text index with support for proximity search can be easily extended to take advantage of semi-structured text, by which we here mean text enriched with XML tags. A generic way is to add all XML tags as special words (that is, recognizable as such), for example, `tag:email` or `tag:subject`. It is then straightforward to extend the proximity operator such that a word is considered close to a particular tag if and only if it occurs between a corresponding tag pair. In CompleteSearch, we use the syntax `..` (two dots) for both normal proximity search (as discussed above) and the search within tags. An example query would be `tag:email..tag:subj..dbworld`, which would retrieve all email messages (tagged as such) mentioning (a word starting with) `dbworld` in their subject line.

This simple trick of using a generalization of "proximity" is, in a certain sense, generic and not CompleteSearch specific. It does not exploit the prefix search and autocompletion mechanism, which is why we do not claim that the CompleteSearch engine is more apt than other (efficient) systems for the purpose of XML retrieval. Still, with this simple trick we can support a subset of the *XPath* query syntax, called NEXI [Trotman 04]. XML support has been added in a similar way to the TopX engine [Theobald 05b], which we will briefly discuss in Section 8.2 in the context of database style retrieval. Note that CompleteSearch permits free mixing of queries using tag information with any of the other query types.

5.7 OR and NOT Operator

Most search engines like Google also support the "advanced" query operators OR and NOT. E.g., the Google query `(web OR internet) search` will return documents containing the word `search` and at least one of the words `internet` and `web`. Similarly, the query `surfing -internet`, where `-` (minus) is used as the symbol for the NOT operator, will return documents containing `surfing` but not containing `internet`. Both of these features we also implemented for the CompleteSearch engine and both of these features also work with prefixes.

To implement the NOT operator, a small modification of the intersection routine was sufficient. This routine takes two lists of document ids (and, optionally, positions) and parallel word ids and, normally, outputs elements from the second list that correspond to document ids contained in both lists. For the NOT variant, we have to output elements from the *first* list which correspond to document ids *not* contained in the second list. Note the following two slight oddities here. First, whereas usually the result set becomes *smaller* as the last prefix gets longer and hence more specific, for the NOT operator the result set becomes *larger* in this case. Second, the completions we display are for the *last but one* prefix, as we prefer to display information about the matching documents (and completions), rather than non-matching ones. Also note that we do not allow the use of the NOT operator for the very first query word, as it is difficult to support efficiently (the result can be enormous, consider `-xyz`) and is also not very sensible to use ("Show me all documents not containing a certain prefix."). The usual (implicit) AND operator requires intersections of lists. The OR operator on the other hand, requires the *merging* of lists, and we also implemented this operation. For words grouped by | (pipe), this operator will be used.

5.8 Phrase and Subword Completion

Another simple yet often useful extension to the basic autocompletion feature is, to consider as potential matches not only the (full) words as they occur in the collection, but also meaningful subwords and phrases. An example involving a subword would be: for the query `normal..vec` we might want to see `eigenvector` as one of the relevant completions.[2] An example involving a phrase would be: for the query `informa` we might want to see the phrase `information retrieval` as one of the relevant completions. The autocompletion according to Definition 1 will automatically provide this feature if only we *add* the corresponding subwords/phrases to the index. For example, for each occurrence of the term "eigenvector", we also add the artificial term "vector:eigenvector" to our index. This term then matches the prefix `vector` and the ":eigenvector" lets us know that we should display the term `eigenvector` to the user. Similarly, for every occurrence of the phrase "information retrieval", we add the term "information_retrieval" to our index, which will be displayed as `information retrieval`.

The problem of *finding* meaningful subwords and phrases to add to the index, is orthogonal to our work. In practice, for the subwords we use a simple greedy approach to find all words of minimal length which appear sufficiently often as subwords of longer words. To identify meaningful phrases, we exploit the fact that such phrases are often written (i) with hyphens, (ii) with underscores or (iii) even as a single long word. These "misspellings" give us a small, high-quality set of word combinations, which we can check for phrase occurrences. Details on this can be found in [Klein-Heyl 07].

[2] Note that we deliberately do not want full substring search, as this would give mostly irrelevant matches for short prefixes.

Chapter 6

Faceted Search

6.1 Introduction

When it comes to finding documents in large collections, the 1-box web search interface approach seems to be predominant. The user enters a couple of keywords and is then presented with a list of matching documents ranked by (ideally) relevance. Such interfaces are very intuitive to use and suffice in many cases. On the other hand, in settings where the focus is more on data exploration than on information retrieval, a hierarchical organization of the documents is often used, allowing the user to browse and drill down into subcategories. Online shopping sites nowadays present the user with both options at the same time: the option of entering queries as usual while also allowing the user to refine his search by drilling down into the matching categories. By pro-actively supporting this kind of query refinement, the user needs to know less about the structure and the items in the database and is spared the experience of over-specifying a request using an "advanced search" form.

In this chapter, we demonstrate how our HYB (see Chapter 4) algorithm and the CompleteSearch engine can be directly applied to obtain both features: pro-active support for both the query formulation (by presenting relevant completions) and for the query refinement through categories (by presenting matching categories). We have built and tested a prototype with these capabilities and verified its practicability in terms of efficiency by experiments with a collection of scientific articles and with the English Wikipedia. To our knowledge, this is the first experimental study of faceted search under the efficiency aspect. Figure 1.1 (in Chapter 1) provides a screenshot of our search engine in action and gives an explanation of its main features. See http://search.mpi-inf.mpg.de/dblp-plus/ for a live demo of this.

6.2 Related Work

The number of websites which offer some kind of faceted search is enormous and spans all information domains. If one loosely defines a faceted search interface as one which, in addition to showing ranked results for keyword queries as usual, also organizes query results by categories, then nowadays almost every online shopping portal offers a faceted search interface (http://www.ebay.com, http://www.amazon.com).

The archetypical, fairly intuitive user interface shows a search box on top, a subset of the most relevant results below and a list of matching categories (usually organized in a hierarchical manner) on the left. These matching categories can then be used for further refining the query. Variations of this interface include extensions to search in the category names themselves. See, e.g., the demos of the Flamenco Project (http://flamenco.berkeley.edu). These type of interfaces are in their simpler form also used in other domains such as medical databases (http://www.medlineplus.gov, provided by Recommind), news archives (http://browse.guardian.co.uk, provided by Endeca) and tagged webpages (http://www.rawsugar.com, provided by RawSugar). Other systems, such as Facetmap, provide no search facility but only allow faceted browsing (http://www.facetmap.com).

A number of content providers support the user in finding relevant terms for his query. Some sites (http://fastsearch.com/search.aspx, http://kayak.com) use a precompiled list of plausible queries similar to Google Suggest (http://www.google.com/webhp?complete=1), from which they display completions to the user. These lists are typically only useful if one is looking for "mainstream" information and for spelling

suggestions. For fully exploring the database's content they are inadequate, as already after a single highly specialized term the list of queries suggested collapses completely. Such interfaces are usually referred to by the term "live search" (see [Rønn-Jensen 06] for a discussion and more related links). In the domain of library search, the AquaBrowser interface (http://aqua.queenslibrary.org) uses data mining, machine translation and spelling corrections to find other related query terms on-the-fly. These are then presented to the user in a star-like graph. However, these terms seem to be often (i) quite general and (ii) unrelated. For the query `information retrieval`, broad and partly irrelevant terms such as "service", "technology" and "freedom" are displayed as associations, along with the supposed spelling variant "Euro-travel" and the supposed translation "enlightenment".[1] Following a fully term-centered approach, it is also plausible to disregard the hierarchical structures for the documents altogether and rather focus on organizing the terms in the collection. Such a taxonomy can then be used to guide the user through the query formulation process, displaying synonyms and other related terms at every step [Binding 04].

For the case of a well-controlled database, whose structural data integrity can be ensured, a query language is presented in [Ross 05], which allows answering faceted database queries in time quadratic in the number of items with linear space complexity. The problem of faceted search can also be viewed more generally as a problem of multidimensional visualization and navigation. To show the relation between two independent dimensions the use of 2-D heat maps was proposed in [Arentz 04] and their benefit is demonstrated by a small user study. Similar ideas are explored in [Shneiderman 00]. Larger user studies to demonstrate the advantages of the Flamenco faceted search user interface have also been conducted in [English 02; Yee 03].

In scenarios, where hierarchical structures or any kind of categorization are not a priori given, it is still possible to apply the paradigm of organizing the result set in a structured way by clustering the results on-the-fly. In [Hearst 06] the relative (dis-)advantages of result clustering and faceted categories are compared. Besides the lack of quality of the resulting clusters (meaning that clusters can be very heterogeneous), even for state-of-the-art systems (http://www.vivisimo.com), other disadvantages include the lack of predictability (a user does not know in advance how his results will be organized) and the diverse mix of the obtained subcluster hierarchies (many facets get mixed when a cluster is broken down).

In our work, we completely factor out the issue of how (hierarchical) labeled categories can be obtained. If one hopes to accumulate a large set of documents, manually labeled, the only viable approach is to have a dedicated community of people all contributing to this project. Such a contribution can be made by manually inserting a web document into a taxonomy, as in the case of the open directory project (http://www.dmoz.org), or by sharing (organized) bookmarks and assigning short tags to a currently visited web site (http://www.rawsugar.com, http://www.flickr.com, http://del.icio.us). The latter approach is often referred to as "social tagging" with the resulting structures being nick-named "folksonomies" (http://en.wikipedia.org/wiki/Folksonomy).

6.3 Faceted Search with Autocompletion

In the previous chapter, we discussed the basic "Autocompletion Search" feature and some of its extensions. The focus of our work in the current chapter is on showing how the algorithm HYB for this "ordinary" autocompletion search problem (see Definition 1), can be extended to include faceted search capabilities. Specifically, we show how the following three features can be obtained:

- a display of matching completions for query terms (occurring in at least one hit) and their use for query refinement,
- a display of matching categories (containing at least one hit) and their use for query refinement, and
- a display of matching category names (starting with the prefix and containing at least one hit for the remaining query) and their use for query reformulation.

The first feature is just the "normal" autocompletion search feature, which was discussed (with extensions) in the previous chapter. Making the changes necessary to provide the other two features will only involve adding special words to our index and will leave the functionality of the basic feature unchanged. The other two features, related to the use of the category information, will be discussed in the following subsections.

[1] It is, however, encouraging to know that our research area is part of the path to eternal bliss.

6.3.1 Finding Categories Containing Matches

To be able to find and display matching categories, we simply add the information about categories to our index by inserting an artificial term, e.g., `cat:living_people`, into a Wikipedia article about a living person before our index structure is built. The same is done for other categories and documents. The colon `:` serves to distinguish artificial words from ordinary words.[2] In the case where we have hierarchical or orthogonal categories, we encode the structure using multiple `:`, such as `cat:author:Donald_Knuth` or `cat:conference:SIGIR`, but for the Wikipedia we worked with a single "flat" categorization.

To process a query, e.g., `pope bene`, we first run the usual autocompletion search query as explained in detail in Chapter 4, also benefiting from extensions discussed in the previous Chapter 5. This gives the set of matching documents in form of a ranked list. Then, in a second step, we run the query `pope bene cat:` *as a regular autocompletion search query*. Due to the way we inserted the artificial terms, the completions for the term `cat:` will now exactly correspond to the list of categories containing a matching document for the query `pope bene`.

The set of matching documents for this artificial query is, on the other hand, not relevant as we do not assume that every document is forcibly tagged with at least one category. In the case of orthogonal taxonomies, where each document is guaranteed to have a label for each facet, the set will be the same as for the query `pope bene`. In general, however, it can be an arbitrary subset.

When these matching categories are presented to the user, he then has the option of refining his search by limiting the search scope to matches within a chosen category. In our example, he might choose the Wikipedia category "Popes". This choice then reduces both the number of matching documents, as well as the relevant completions for `bene`. Figure 1.1 (in Chapter 1) shows matching categories listed as "Refine by" options. Here the name of an author and a year of publication are also considered categories.

In our user interface, the category selection is done by simply clicking on a relevant category. This follows the standard conventions for providing such a feature and is implemented in a similar manner in all major search interfaces for faceted search (`http://www.rawsugar.com`, `http://flamenco.berkeley.edu`, `http://www.ebay.com`, `http://www.amazon.com`, ...).

Observe that selecting a particular category is, from the point of view of our algorithm, the same as entering an additional query term, where the query term in this case is an artificial term. Just as there is no limit on the number of query terms the user can enter, there is no limit on the number of categories which can be selected during the iterative query refinement process.

In scenarios where the total number of matching categories can be very large, as is the case for the Wikipedia collection, we only present the most relevant ones to the user. Again, for this we can employ the very same ranking mechanism which we already used to select the most promising completions (see Section 5.4).

To ensure efficient query processing, all artificial terms of the form `cat:*` should be in the same block for HYB. That is, our data structure is built such that we have the sorted list of documents containing at least one of these special words precomputed. Given such a choice of blocks, we can then answer a query of the form `pope bene cat:` above with a single list intersection. Exactly for these cases, where the list of potential categories (i.e., "words" starting with `cat`) is large, HYB shows the best performance compared to an approach based on the inverted index, which would have to iterate over these candidates.

6.3.2 Finding Matching Category Names

In some cases, the user's intention for the query `pope bene` might be of a different nature: Maybe when typing such a query the user is looking for *categories* starting with `bene` and containing a document with the term `pope`. For the Wikipedia such matching categories would include "Benedictines" or even "Benefit_albums". The "Categories matching" field in Figure 1.1 (in Chapter 1) contains an example where a particular author is suggested as a matching category.

Using exactly the same setup as before with the same artificial words included in the index, such matching completions can be found by answering the autocompletion query `pope cat:bene`.

Again, the user can now choose from any of these matching categories. But, different from before, a selection of a particular category does not correspond to a query refinement with respect to the base query

[2] This trick will be used repeatedly in the following chapters.

`pope bene`, as now one query word (`bene`) gets *replaced* by another (`cat:bene`) whereas before a query word was *added*.

The selection of such a desired category is again done by clicking on the name of the category in our user interface. This feature is less common but, if present, is usually provided via the same mechanism, as in the Flamenco System (`http://flamenco.berkeley.edu`).

If the blocks for HYB are chosen optimally as for the feature above, then we can again answer a query of the form `pope cat:bene` above with a single list intersection and avoid any merge operations. In practice, these types of query will be easier to answer than the query types of the preceding subsection, as the relevant word range (all category names starting with `bene`) is narrower than before (where *all* category names had to be considered).

6.4 Experiments

We integrated the faceted search feature into our CompleteSearch engine, whose core is written in C++. The experiments were done on an Opteron dual 2.4 GHz processor machine with 8 GB of RAM, with the (compressed) index on disk. In our experiments, we measured the time for processing both the "normal" and the derived autocompletion search queries, as described in Sections 6.3.1, and 6.3.2. We did *not* measure the time for the transmission of the query and the results over the network.

6.4.1 Collections and Queries

Our first data set, DBLP, consists of 11,685 scientific articles listed in DBLP (`http://dblp.uni-trier.de`), both the full text and the DBLP meta data, including information about authors, conferences and year of publication. Note that all of the three facets (author, conference, year) are "flat" in the sense that they do not come with a hierarchy.

Our second data set, WIKIPEDIA, consists of the full set of 2,172,832 articles of the May 2006 dump of the English Wikipedia (`http://en.wikipedia.org`). As meta data for this collection, we took the Category information which the Wikipedia articles themselves provide (at the bottom). This information is much more diverse than for DBLP: some articles carry a dozen of different category labels, about half of the articles, especially if they have been recently added or are navigational pages, are listed under no category at all.

For both collections we generated 500 standard keyword queries with a realistic distribution of query length (short queries are more common) and keyword selection. As for our other experiments, where queries were generated artificially, we ensured that content-bearing words were more common in the queries than very frequent words by using $tf - idf$ based sampling. See Section 4.5.2 for details. Each query was then "typed" letter by letter (beginning with a 3-letter prefix for each query word), resulting in a chain of autocompletion queries. For example, the keyword query `pope benedict` gives rise to the 8 autocompletion queries `pop`, `pope`, `pope ben`, `pope bene`, ..., `pope benedict`. Like this we obtained 6024 autocompletion search queries for DBLP, and 5320 autocompletion search queries for WIKIPEDIA.

From each such autocompletion search query, then three queries were derived: the "normal" query itself (to find relevant documents and word completions as discussed in the previous Chapter 5), the query with the query word prefix `cat:` added at the end (to find matching categories as described in Section 6.3.1), and the query with the last query word prefixed by `cat:` (to find matching category names, as described in Section 6.3.2).

6.4.2 Results

Table 6.4.2 shows the average running times per query on both DBLP and WIKIPEDIA, with a breakdown for the three sub-queries described above. Three main observations are to be made. First, for both collections, queries are processed in a fraction of a second, which yields the desired interactive behavior. Second, the query processing time is dominated by the second type of subqueries. This is easy to understand, since for the second type of query, we have to screen each matching document for its category labels. This is a work-intensive task, but as explained in Section 6.3.1, this is exactly the kind of queries where our HYB data structure shines, and improves over the standard inverted-index based approach by an order of magnitude (see Chapter 4 for details).

Collection	DBLP	WIKIPEDIA
Average time	24 millisecs	341 millisecs
- ordinary	6 millisecs	53 millisecs
- categories	17 millisecs	274 millisecs
- cat. names	1 millisecs	14 millisecs

Table 6.1: Average running times for the faceted autocompletion queries on DBLP and WIKIPEDIA, with a breakdown with respect to the three subqueries discussed in Section 6.3.

Third, the last type of sub-queries hardly takes any processing time, since for most autocompletion queries, the last query word does not match any category name at all (in which case the cost for this sub-query is zero).

Chapter 7

Synonym Search

7.1 Introduction

One of the central problems of keyword-based search is that often there is not a unique set of keywords to identify a topic. Sometimes words referring to the same concept are merely morphological variants (compute, computer, computing) which can be identified by stemming [Porter 80], i.e., reducing words to their "normalized" form, or even less trouble-some by using prefix search, as supported by the CompleteSearch engine, where it is left up to the user to decide whether he effectively wants to use stemming (and only type a shorter prefix) or not (and type the full word).

But almost always there are also closely related terms, e.g. car, automobile or vehicle, which are not just morphological variants. Adding such related terms to the query in order to increase recall, is an old and much-researched technique commonly known as *query expansion*. A recent, very good overview is provided in [Billerbeck 05]. In this chapter, we show how we can use the CompleteSearch engine for query expansion. Our work is different from the traditional approach in two respects.

Efficiency The standard way to implement query expansion is to replace each query word by the *disjunction (OR)* of its related terms, and then *merge* the individual inverted lists [Billerbeck 05]. Since each step of such a merge is logarithmic in the number of lists, expansion is typically limited to a few important words. Sophisticated top-k techniques, like in [Theobald 05a], try to prune expansion words which are unlikely to lead to an improved ranking.

Interactivity & context-sensitivity Our feature interactively suggests words related to the word currently being typed, and the suggestions are ranked by their ability to lead to good hits together with the preceding part of the query. For example, for the query `russia metal` from Figure 7.1, `aluminum` is high up in the list of words related to `metal`, because there are many news articles about aluminum production in Russia. Similar interactive features have been discussed in the literature [Fonseca 05], but with the focus on effectiveness and not on efficiency.

We do not claim novelty for any of these individual points (efficient, interactive, context-sensitive). However, we have not seen them presented in combination, and we want to stress the simplicity with which we realize this feature here, using CompleteSearch's efficient prefix completion mechanism.

7.2 Query Expansion via Prefix Completion

Given knowledge about related terms, we could, when the user is typing `rent automo`, suggest the term "car" to him, assuming that the query `rent car` would lead to a hit.

One way to obtain such a feature would be as follows. In a first phase, go through each completion of `automo` (e.g., automobile, automotive, automower) and for each such completion get the list of synonyms (e.g., car, vehicle, wagon). In a second phase, for each such synonym we could launch the corresponding query such as `rent car`, `rent vehicle` or `rent wagon`. The subset of such queries, which leads to a hit, gives us the "completions" to display to the user.

The obvious problem with this approach is that a single initial query can now lead to several queries in the second phase, not exploiting HYB's strength to deal with prefix *ranges*. This can be avoided by introducing

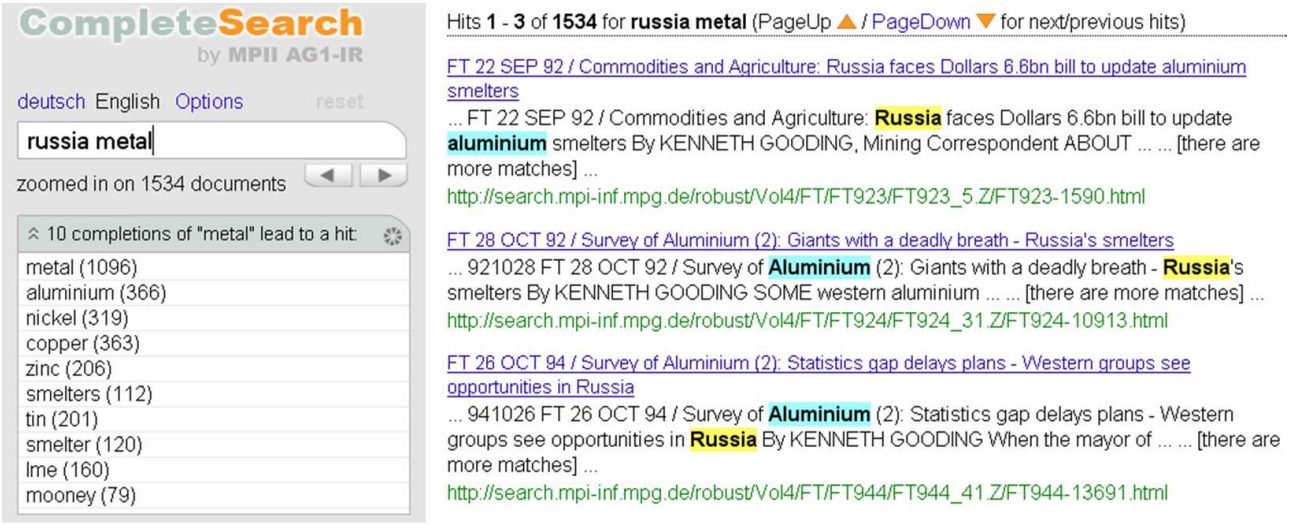

Figure 7.1: The proposed feature, integrated into the CompleteSearch engine. The box under the search field shows words related to the last query word, metal in this case, that would lead to good hits, and to the right the best such hits are displayed. Both related words and hits are updated automatically after each keystroke.

artificial prefixes corresponding to groups of synonyms. Then the desired feature becomes a matter of a few prefix completion operations, which are efficiently supported by HYB. We realize this as follows:

0. As input, assume we have *clusters* of related terms. These clusters may overlap, for example the word `case` may be in one cluster together with `cover`, `shell`, etc. as well as in another cluster together with `box`, `chest`, etc.
1. For each occurrence of a term `<t>` that occurs in a cluster with id `<id>`, add the artificial term `s:<id>:<t>`, in the same document and at exactly the same position.

   (Document position is important for proximity and phrase search. We remark that the HYB index does not mind several terms at the same position in the same document.)

   Also add, but only *once* for each such term and in a special document that is used for no other purpose, the artificial term `s:<t>:<id>`. The number of words in this document is just the total size of all clusters.
2. For a given query `<q1> ... <ql> <p>`, we then realize our feature via one or two prefix completion operations as follows: We first check whether `<q1> ... <ql> s:<p>` has a unique completion of the form `s:<t>:<id>`. If so, this completion gives us the id `<id>` of the cluster containing `<t>`. Then the query `<q1> ... <ql> s:<id>:` gives us the desired completions and hits.

   (Note that the part `<q1> ... <ql>` is evaluated only once, just after its last letter being typed, and stored by a cache-like mechanism from then on, see Section 11.8 for details.)

The terms added under point 2. above do, of course, lead to an increase in the total space usage. By how much depends on (i) for how many words we have synonymy information, (ii) how often these words occur in the corpus and (iii) in how many synsets these words are contained. It does, however, not depend on the sizes of the synsets. If we chose the straight-forwards approach of introducing all of the synset's elements (e.g, car, automobile, vehicle, ...) for every occurrence of a word (e.g., car) from the synset, then the space blowup could become dramatic.

7.3 Term Clusters

We implemented and tested our feature for two collections: the TREC Robust collection (1.5 GB, 556,078 documents), and the English Wikipedia (8 GB, 2,863,234 documents). For each of the collections we used a different method to derive the clusters of related terms, one unsupervised and one supervised. The following two subsections give details on each of the methods. The results of the experiments are given in Section 7.4.

7.3.1 Unsupervised Approach - Spectral Method

For the Robust collection, we used a completely unsupervised approach based on the technique from [Bast 05], as follows.

1. Since Eigenvector computations on large matrices are very expensive, we first removed common stopwords, and restricted ourselves to a set of frequent nouns, which we identified using the TnT tagger [Brants 00]. We then ran Porter's stemming[Porter 80] algorithm on these nouns and got a set of 10,098 terms to work with.

2. We obtained a set of related term pairs from these 10,098 terms using the smoothness test described in [Bast 05]. We used a high smoothness threshold to select term pairs to ensure that no two unrelated (or not closely related) terms qualify as related terms. Figure 7.2 shows the kind of term-term relations extracted this way.

3. We used the Markov Clustering Algorithm (MCL) algorithm[1] from [van Dongen 00] to derive clusters from the list of term pairs, as required by our approach.

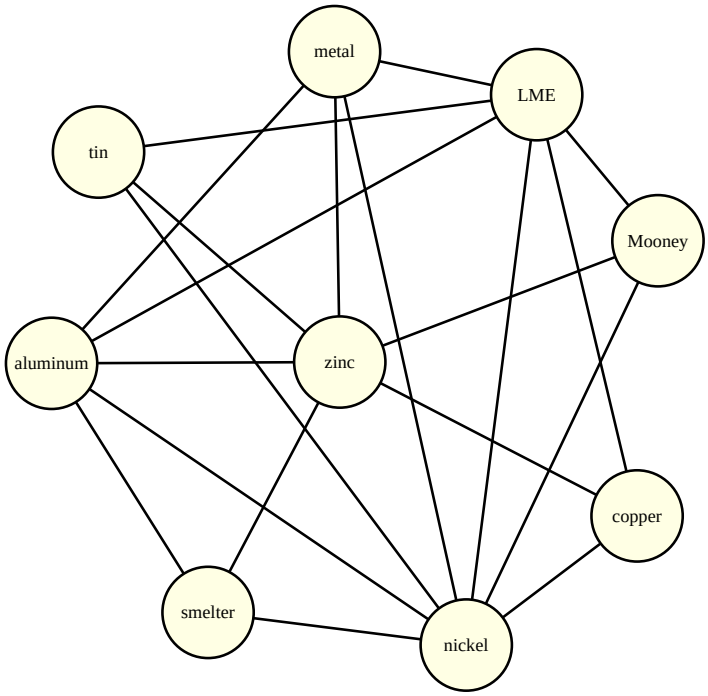

Figure 7.2: One of the clusters of related terms automatically obtained from the Robust collection. Edges present in the graph denote term-term relations found by the smoothness test from [Bast 05]. The cluster itself was then found using the clustering algorithm from [van Dongen 00]. Indeed, all the terms in the cluster are closely related: most of them are different metals, LME stands for London Metal Exchange, and Richard Mooney is the author of several articles regarding the general topic of metal.

Note that this approach makes the result of this unsupervised learning algorithm, and its effect on the search results, completely transparent to the user. In contrast, methods in the spirit of latent semantic indexing [Deerwester 90] are often criticized for their incomprehensibility on the side of the user concerning why a certain document show up high in the ranking. It would be interesting to verify the significance of this difference in a user study.

[1] http://micans.org/mcl

7.3.2 Supervised Approach - WordNet

For the Wikipedia, we made a straightforward use of WordNet [Fellbaum 98] to obtain clusters of related terms. Namely, we put two words that occur somewhere in Wikipedia in the same cluster if and only if they share the same most frequent synset. E.g., for the term "car" the synset corresponding to "auto", "automobile", "machine" and "motorcar" was used, but *not* the ones corresponding to "railcar" or "gondola". Table 7.1 shows all synsets for the term "car". This heuristic leads to only about 30% more tokens in the index. Using *all* synets, would spoil both efficiency and usefulness of our feature.

1. car, auto, automobile, machine, motorcar (a motor vehicle with four wheels; usually propelled by an internal combustion engine) "he needs a car to get to work"
2. car, railcar, railway car, railroad car (a wheeled vehicle adapted to the rails of railroad) "three cars had jumped the rails"
3. car, gondola (the compartment that is suspended from an airship and that carries personnel and the cargo and the power plant)
4. car, elevator car (where passengers ride up and down) "the car was on the top floor"
5. cable car, car (a conveyance for passengers or freight on a cable railway) "they took a cable car to the top of the mountain"

Table 7.1: A complete list of the WordNet synsets for the noun "car". For our experiments, we assigned each word only to its most frequent synset, so for "car" we used the first set in the list above. The term "machine" would in the end not be used as a synonym for car, as its most frequent synset refers to a different concept.

Furthermore, we only used single terms and ignored compound nouns in open form ("lawn tennis"), as we build our index for individual terms.[2] The descriptions and the example phrases for the synsets were also not used, as they do not explicitly contain any synonymy information.

7.4 Experiments

We integrated the described feature with our CompleteSearch engine, and measured its efficiency on two query sets. The first query set is derived from the 200 "old"[3] queries (topics 301-450 and 601-650) of the TREC Robust Track in 2004 [Voorhees 04]. For the second query set, we started with 100 random queries, generated as follows: For each query, we picked a random document with uniform probability and sampled 1 to 5 terms according to their tf-idf values from it, i.e., each term had a probability of being sampled proportional to $tf \cdot \log(n/df)$, where tf is the number of occurrences in the given document, n is the total number of documents and df is the number of documents containing this particular term. The terms were required to lie within a small enough window as to ensure at least one proximity hit. The number of query terms for these queries was chosen with a mean of 2.2 and a median of 2, which are realistic values for web search queries [Spink 02].

For both query sets, these raw queries were then "typed" from left to right, using a minimal prefix length of 3. So the raw query "cult lifestyles" would yield the autocompletion queries `cul`, `cult`, `cult lif`, `cult life` and so on. Additionally, whenever for a prefix <p> the query `s:`<p> led to a unique term cluster with id <id>, we added an OR (for which we use the "|") with the prefix `s:`<id>`:`. E.g., one autocompletion query in the sequence for "airport security" is `airport|s:399: secu|s:385:`.

All experiments were run on a machine with two 2.8 GHz AMD Opteron processors (two cores each, but only one of them used per run), with 16 GB of main memory, operating in 32-bit mode, running Linux.

Table 7.2 shows that, by using the term clusters, the average processing time increases by roughly 50% (but not more) with respect to queries without synonymy information, and it is still well within the limits of interactivity. Somewhat surprisingly, the maximum processing time is *lower* for the queries *with* synonymy information. This is because the queries which take the longest to process are those with a very unspecific last query word, for example, `cont`. Such words tend to have more than one completion for which synonymy

[2] Inclusion of such compound nouns is theoretically possible, but it was not implemented for this study.
[3] They had been used in previous years for TREC.

Table 7.2: Breakdown of processing times for both of our query sets. For "normal" queries there was no synonymy information to be used.

Query set	Average	90%-tile	99%-tile	Max
Robust (all)	32 ms	90 ms	375 ms	970 ms
- normal	22 ms	55 ms	329 ms	970 ms
- synonyms	57 ms	129 ms	385 ms	655 ms
Wikipedia (all)	64 ms	238 ms	614 ms	1218 ms
- normal	42 ms	128 ms	569 ms	1218 ms
- synonyms	35 ms	356 ms	799 ms	841 ms

information is available, and in that case our interface, as described above, does not show any related terms, but only syntactic completions.

Chapter 8

DB-style Search

8.1 Introduction

In this chapter, we show how we can use CompleteSearch to process a wide class of database-style queries. First, we discuss other work related to the topic of "bridging the gap between DB and IR" in the following Section 8.2. Then, in Section 8.3 we explain that, by creating special documents with the appropriate artificial words, we can put an arbitrary relational table into a form, where it can be processed by the CompleteSearch engine. Section 8.4 gives examples of the types of query that the CompleteSearch engine can, with these artificial words added, process and how this is done. Interestingly, we easily obtain support for a mix of key word and db-style queries, thus taking a step toward DB+IR integration. Besides CompleteSearch's autocompletion feature, the capability to process *joins* will be central for this. Section 8.5 discusses how arbitrary joins can be processed. Finally, in Section 8.6, we prove experimentally that this added functionality does not impede CompleteSearch's ability to efficiently handle even advanced queries.

8.2 Related Work

The QUIQ engine [Kabra 03] is another recent attempt to integrate IR and DB functionality into a single system in a uniform manner. QUIQ is built on top of a DBMS, partly motivated by their focus on *dynamic updates* (to which we give only relatively little attention, see Section 12.2.3). Like CompleteSearch, QUIQ makes extensive use of the idea to "map non-text data to pseudo-keywords that cannot be confused with actual keywords of text", and for the case of CompleteSearch this will be discussed in the next section.

The *TopX* engine, developed by our colleagues at MPII [Theobald 05b], combines search in semi-structured (XML) data with techniques for top-k retrieval, with a strong focus on the latter. As explained in Section 5.6, CompleteSearch supports exactly the same subset of XPath queries as TopX. Like QUIQ, TopX is built on top of an off-the-shelf DBMS (Oracle).

The *HySpirit* system [Fuhr 98] was designed for "hypermedia retrieval integrating concepts from information retrieval and deductive databases." The system is based on a probabilistic model of Datalog. Like CompleteSearch, it can combine ranked retrieval with database queries. Like QUIQ and TopX, HySpirit is built on top of a DBMS.

Our work on CompleteSearch addresses some of the issues and challenges raised in a recent overview paper by Chaudhuri, Ramakrishnan, and Weikum [Chaudhuri 05]. Our central completion mechanism might be viewed as an instance of the "storage-level core system with RISC[1]-style functionality" argued for in [Chaudhuri 05]. We certainly agree with their point of view that an integrated IR&DB (or DB&IR) system should *not* be built on top of an SQL-engine or a vanilla B-tree implementation, for reasons of efficiency. This is discussed in more detail in Chapter 11 (in particular see Table 11.1). Flexible scoring and ranking and high-performance query processing, the first two items on the requirement list of [Chaudhuri 05], are at the core of the design of CompleteSearch. See Section 5.4 for details on the scoring mechanism used by the CompleteSearch engine.

[1] Reduced Instruction Set Computer. See [Chaudhuri 00] for a discussion of the relation of this concept to DB-system design.

For a more thorough overview of the area of IR&DB-integration, we refer the reader to the SIGMOD'05 panel discussion [Amer-Yahia 05], in particular its references. A classification of existing schemes according to criteria such as integration architecture and general approach is attempted in [Raghavan 01]. In the following sections we discuss our approach to combine basic IR and DB functionality.

8.3 Putting Data Tables into Document Form

Consider a collection of computer science articles, where for each article we have its conference, the authors and the year of publication. Let's, for now, assume that we have only this metadata and no full text. How can we preprocess this data, such that it is in a form "digestible" by our CompleteSearch engine?

We do this by transforming the corresponding table, with the schema (conference, author, year), as follows: we add special words of the from <category name>:<category instance> to a (new) document pertaining to the article. The colon simply serves to distinguish these words from ordinary words. For example, we might add conference:vldb, or author:jon_kleinberg, or year:2006. Generally, given any table with attributes *attr_1* up to *attr_n* we add the attribute-value pairs as attr_1:<val_1> up to attr_n:<val_n>, where <val_i> is the entry for attribute *i*. All the words of this document will then correspond to a *row* of the relational table where the doc id acts as an implicit, unique ID for each row. All the words corresponding to a special prefix, e.g., conference:, will correspond to a *column* of this table. If we have more than one table, we also add the table name, say *ABC*, as table:ABC to each such document.

Recall from Chapter 4 that, when the block boundaries are chosen appropriately, HYB stores the word-in-document pairs for such a special prefix contiguously in memory, just by the way it works. This is known in the database world as *column store* [Stonebraker 05], which is generally preferred in systems optimized for read, rather than for write access. With such a layout one can efficiently obtain all the information in a particular column. If we later also want to support queries for the full text, we simple add the whole text to the same document with the artificial words.

8.4 Supported DB-style and Mixed Queries

Consider the *two* queries conference:sigir author: and conference:sigmod author: . According to Definition 1, the first query produces a list of authors (as completions) who have published at SIGIR, along with the corresponding publications (as hits). Similarly, the second query produces a list of authors who have published at SIGMOD, along with the corresponding publications. Now let us us intersect the two lists of authors, that is, the lists of (ids of) *completions* of the two queries. Details will be given in the following section. Note the duality to the archetypical search engine operation of intersecting lists of (ids of) *documents*. The intersection of the two lists of completions gives us the lists of all authors, which have published at both SIGIR and SIGMOD, and the two lists of documents provide the witnesses of these facts. That is, we have effectively computed a *self-join* on the table which he have implicitly created by the addition of the special words. In Section 8.5, we explain how to generalize this to arbitrary joins. Note that the information required to process this kind of query is spread over several documents, which is something standard IR-style keyword search cannot handle. For example, the query conference:sigir conference:sigmod author: would not match any document, because no document is a SIGIR paper and a SIGMOD paper at the same time.

If we prepended ir db integration to the two queries above, we would obtain the join table restricted to documents matching this query, and we would obtain a list of authors which have published at both SIGIR and SIGMOD about the topic of IR&DB integration. This is a first example of how CompleteSearch can combine IR-style with DB-style querying and what kind of more advanced queries we can process, given the join capability. Our semantic search engine, built on top of CompleteSearch and presented in Chapter 9, will make heavy use of such combined queries and of the join operation.

8.5 General DB-style Joins

In this section, we explain in detail how we can process arbitrary joins with the CompleteSearch engine. Suppose that we are given two tables *ABC* and *XZY* and, as a starting point, that we would like to compute their

inner join for the attribute *attr_k*. We then launch the queries `table:ABC attr_k:` and `table:XYZ attr_k:` as prefix search and completion queries according to Definition 1. For the two result sets, we then have to intersect the lists of matching *completions* (not documents). Unfortunately, such lists of word ids cannot be intersected just as easily as the document ids, because the lists of word ids are *unsorted*. One approach, called merge join, would be to first sort the lists by word id and then intersect. An alternative would be to use a hash join (computing the list of word ids that occur in both lists via hashing). As the word ids always come from a small *range*, we can compute the intersection efficiently via a realization of a hash join with two bit vectors. We experimented with both variants (see Section 8.6 for details) and the hash join was generally faster by a factor of 3-5. A general discussion about join processing can be found in [Mishra 92].

Once we have computed this intersection, the elements contained in both lists are exactly the matching attribute-value pairs for the join attribute. To obtain the corresponding rows of the join result table efficiently, we profit from the fact that whenever we are intersecting lists of word ids with HYB, we are actually handling *pairs*. I.e., we also have the corresponding document id at hand (and vice versa when we are intersecting lists of document ids). This way we can easily obtain the corresponding document ids, which correspond to the matching rows in both tables. By slightly modifying the intersection routine to output NULL when a word id is present in only one of the two lists, we can use the same procedure to compute left, right or outer joins as well.

Since the special words for a particular attribute *k* (starting with the prefix `attr_k:`) of any such table share a common prefix, they will be stored in consecutive locations by HYB, and will either form their own block, or be part of a single block. This allows for an efficient processing of join queries. Note that the complex problem of *join ordering* [Swami 89; Steinbrunn 97] does not occur in our interactive setting, because the fact that we want results after every keystroke demands an evaluation of the query in a strict order from left to right.

8.6 Experiments

To test, if the join operation becomes the bottleneck for processing advanced autocompletion search queries, we experimented with such advanced queries on two different collections.

The first collection (DBLP) consists of about 20,000 scientific articles. For each of these articles the index contains both the full text and meta data from the DBLP data set[2]. The meta data we used comprises information for 24,028 authors, for 26 conferences, and for 33 years of publication.[3] The queries we used were of the type discussed at the beginning of Section 8.4. Namely, for each of the 325 pairs of (distinct) conferences we asked (i) for a list of authors, who have published in both conferences, as well as (ii) the list of their publications.

For our second collection (Wiki+Yago) we combined the Wikipedia with the Yago ontology [Suchanek 07], which makes it possible to answer certain more *semantic* queries. This collection and its applications will be discussed in detail in the following Chapter 9. For the queries (which are the "hard ontology queries" from Section 9.9.1) we used 1000 persons, present in both the Wikipedia and the ontology, and for each such person we asked for the death dates of all persons that were born in the same year as the given person. Processing these queries involves a join on the (common) birth years. The data comprises a total of 2.9 million documents and information about the birth years of 165,000 people.

The experiments were done on an Opteron dual 2.4 GHz processor machine with 8 GB of RAM. The index resided in main memory (disk cache). While running the queries we made extensive use of our result caching mechanism (see Section 11.8) to avoid, e.g., that the information for the prefix `author:` has to be read multiple times from disk. The time it takes to transmit the results over the network was not included in the measurements.

For the DBLP collection, all the queries were, due to the collection size, easy to process with an average processing time of a mere 2.2 milliseconds (using hash joins). For the Wiki+Yago collection, the average processing time for the advanced queries was 65 milliseconds (also using hash joins). However, the absolute processing time was not under investigation here. More interestingly, the hash join turned out to be 4 – 6 times faster than the merge join and did not make up more than 30% of the average total processing time.[4] Details are given in Table 8.1. Note that the relative processing time would decrease further if the index had resided on

[2] http://www.informatik.uni-trier.de/~ley/db/index.html

[3] This collection is a more recent version of the DBLP collection already used for the experiments in Chapter 6.

[4] An experimental evaluation of the HYB data structure (underlying the CompleteSearch engine) for the basic prefix search and completion queries was given in Section 4.5.

Collection	Merge join	Hash join
DBLP	30% (0.9 ms)	9% (0.2 ms)
Wiki+Yago	71% (121 ms)	30% (20 ms)

Table 8.1: Query processing times for advanced queries involving a join operation. The percentage refers to the percentage of the average total processing time, when the particular join implementation is used. The time in parentheses is the average time spent in the join operation.

disk (and we had ensured that no disk caching was performed by the operating system) and if we had not used any result caching mechanism. Further experiments for other advanced queries involving the join capability are given at the end of Chapter 9.

Chapter 9

Semantic Search

9.1 Introduction

In the previous Chapter 8, we showed that, by adding a join functionality and by augmenting the corpus with the appropriate special keywords, CompleteSearch could do much more than only ranked keyword (or rather prefix) search. Here, we take these ideas to the next level by using the CompleteSearch engine to handle queries, which are of a more *semantic* nature. For example, consider the query "which musicians are associated with The Beatles". This requires a search not for the literal *word* musician, but rather for *instances of the class* it denotes.

Already this simple query highlights two of the main challenges of semantic search: (1) obtain the necessary semantic information, in this case, identify each occurrence of a musician in the given text collection; and (2) make that information searchable in a convenient and efficient way.

Concerning (1), there are actually two problems hiding in here. First, we need to find out which entities are musicians. Second, given such a list of musicians (including "Elvis Presley" and "Britney Spears"), we need to identify occurrences of these entities in the corpus, which might look like "king of rock" or "teen pop star". For our concrete application of the CompleteSearch engine to the Wikipedia, we used the YAGO ontology [Suchanek 07] to address the first issue and the link structure present in Wikipedia for the second issue. Details will be given in the following sections.

The focus of this chapter, however, is on (2): given semantic information, make it searchable fast and conveniently. The main problem here is that standard IR data structures like the inverted index (see Section 2.2) do not provide the necessary functionality. All research prototypes that we know of either use an ad-hoc extension of the inverted index or they are built on top of a general-purpose database management system. In either case, they do not scale well for retrieval tasks on large collections: they either use a lot of space, or they are slow, and sometimes both. This is discussed more in Section 9.3.

The remainder of this Chapter is structured as follows. Section 9.2 gives a short summary of the results presented in this section. The following Section 9.3 discusses a number of other systems which also combine full-text and ontology search. Section 9.4 will provide details on our query engine. Section 9.5 will describe how we add the ontology as artificial words to the corpus. Section 9.6 will describe how entity recognition gives us combined full-text and ontology search. In Section 9.7 we prove that with this approach can handle all basic SPARQL graph-pattern queries, and in Section 9.8 we describe an intuitive user interface to the low-level query engine. In Section 9.9, we describe our experiments with regard to both efficiency and search quality.

9.2 Results

In this Chapter, we show how to use the CompleteSearch engine to obtain a modular system for highly efficient combined full-text and ontology search. Besides the CompleteSearch engine itself, such a system requires (i) an ontology, (ii) an entity recognizer, and (iii) a special user interface. The job of the ontology is to provide us with basic knowledge about existing entities ("John Lennon is a musician"). The entity recognizer then "applies" this knowledge to the corpus by assigning words or phrases to entities they refer to. Finally, the user interface maps simple queries input by the user to the only two basic operations the CompleteSearch engine supports: *prefix search with autocompletion* and *join*. The main challenge in the design of such a system was

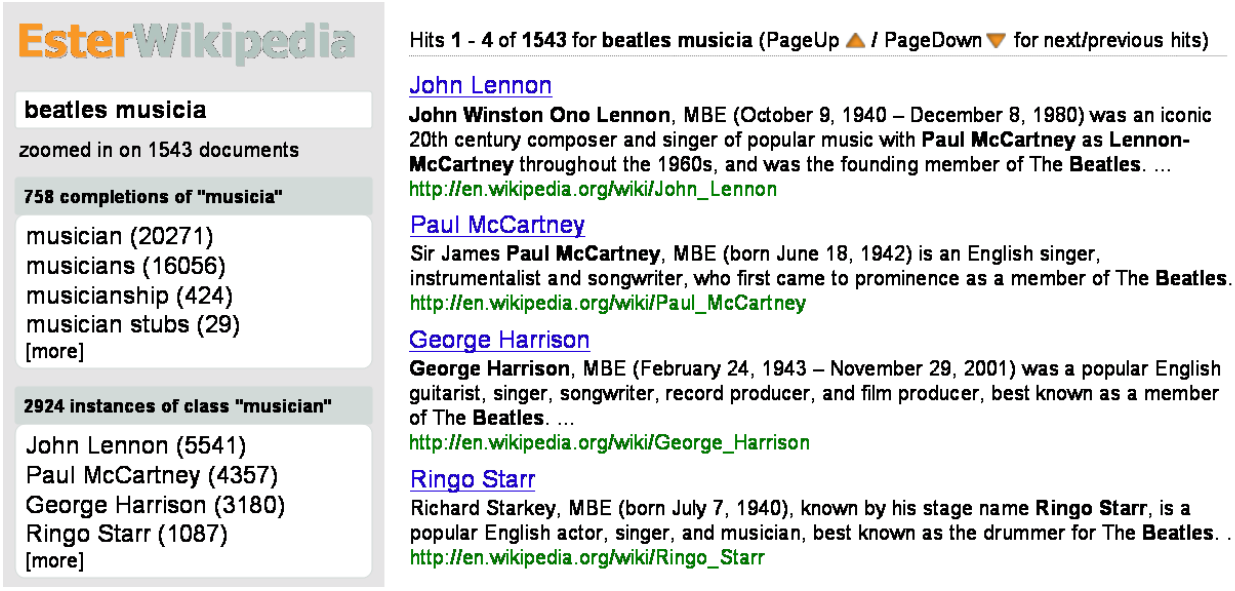

Figure 9.1: A screenshot of our CompleteSearch engine for the query `beatles musicia` searching the English Wikipedia. As for the other applications, the list of completions and hits is updated automatically and instantly after each keystroke, and the number in parentheses after each completion is the number of hits that would be obtained for that particular completion. The upper box suggests words and phrases that start with `musicia` and that occur together with the word `beatles`. The lower box suggests *instances of musicians* that occur together with the word `beatles`. In fact, fast processing of this apparently simple query requires the whole complexity of our system in the background: prefix queries, join queries, entity recognition, and ontological knowledge; see Section 9.6. Our interactive and proactive (suggest-as-you-type) user interface hides this complexity from the user as much as possible. See Section 9.8 for other types of queries which the CompleteSearch engine, when the appropriate information is added to its index, can handle in a similar fashion.

to map the knowledge from the ontology and the output of the entity recognizer to artificial words such that complex semantic queries can be processed by mere prefix search and join operations. We show how this can be done efficiently for all basic SPARQL graph pattern queries.

As a proof of concept, we have implemented the whole system with an entity recognizer following insights from [Dill 03], and a user interface, with a similar look to, but (partly) different functionality from the one of our "standard" CompleteSearch engine. The ontological knowledge was provided by the YAGO ontology [Suchanek 07]. For this whole system, we conducted a variety of experiments regarding both efficiency and search quality. The key novelties are as follows:

Scalability By building on the CompleteSearch engine, we can process complex semantic queries extremely fast, with a very compact index. On the Wikipedia corpus, which has about 3 million documents, together with the YAGO ontology, which has about 2.5 million facts, we achieve processing times of a fraction of a second for a variety of complex queries, with an index size of just about 4 GB. This comes close to state of the art full-text search with respect to both query processing time and index size, but with much enhanced querying capabilities; see Section 9.9. Compared to systems with comparable querying capabilities, this is faster by up to two orders of magnitude; see Section 9.3.

Modularity Each of our system's components is easily exchangeable. We could use another data structure for the query engine, as long as it can process the two required basic operations: prefix search and join. For example, one could replace the underlying HYB data structure (Chapter 4 by AutoTree (Chapter 3). Similarly, the only requirement concerning the user interface is that it translates whatever input it gets from the user to these two basic operations. Since the task of the entity recognizer is independent of the indexing and query processing, both it and the ontology used are easily exchangeable, too.

Queries supported We show how our system can solve arbitrary basic SPARQL graph-pattern queries, by reducing them to the basic operations of prefix search and join. If the SPARQL query is a tree with m edges,

we can show that at most $4m$ basic operations are needed. SPARQL is one of the standard query languages for ontologies, and a query is essentially a labeled graph to be matched against the ontology graph; see Section 9.7.

User interface We carefully designed the user interface, so that it is intuitive (easy to use), interactive (short response time) and proactive (automatically trying out "sensible" interpretations of the query). For example, when a user has typed `beatles musician`, the system will give instant feedback that there is semantic information on musicians, and it will execute, in addition to an ordinary full-text query, a query searching for instances of that class (in the context of the other parts of the query), and it will show the best hits for either query. See Figure 9.1 for a screenshot of the system in action for that query. Our user interface builds on experience we gained from the extension of CompleteSearch to also support faceted search (see Chapter 6). Note that by "user interface" we do not so much refer to graphical design and layout issues, but really to the issue of an adequate *interface* between the user, who should not be bothered with complicated syntax, and the system in the background, which only understands a low-level "language".

Entity recognition For the entity recognition component, we implemented a general-purpose semi-supervised algorithm following ideas and insights from [Dill 03]. For our Wikipedia application, we took the links between Wikipedia pages as training data. We achieve a precision of about 90%, which is similar to what is reported in [Dill 03] for a collection of 264 million web pages.

9.3 Related Work

There are still relatively few systems that explicitly combine full-text and ontology search. In none of the systems we know, efficiency was a primary design goal, and performance measurements are often available only as anecdotal evidence. A typical example is the recent system of [Castells 07], which supports essentially the same class of combined semantic and full-text queries as our system. The authors report an "average informally observed response time on a standard professional desktop computer [of] below 30 seconds" on a corpus (CNN news) with 145,316 documents and an ontology (KIM) with 465,848 facts; index size is not reported. This has to be contrasted with the subsecond query times achieved using our system, based on the CompleteSearch engine, on the 2.8 million document Wikipedia, with a provably compact index.

The powerful XQuery language can be used for the kind of queries we consider here. However, experiments (not reported in detail here) with the currently fastest engine, MonetDB/XQuery [Boncz 06], have shown processing times that are two to three orders of magnitude slower than what we achieve using our system. Another alternative is the XML fragment search of [Carmel 03], which deals with a subset of XQuery and which can be used for some, though not all of our queries. While most semantic search engines are built on top of a database management system, with queries being translated into SQL, the engine of [Carmel 03] builds on an inverted index. Similar to the CompleteSearch engine, prefix search on artificial words is used, but without an efficient implementation, and neither query times nor index size are reported.

Völkel et al., in their "Semantic Wikipedia" paper [Völkel 06], propose a semantic Wiki engine which makes it easy for users to add semantic information while creating or editing Wiki pages. This approach, if accepted by the community, will combine semantic information with full text information, but it will not provide a means of searching this information efficiently.

The general idea of enhancing full-text search by the addition of artificial words is, of course, not new. In the QUIQ system [Kabra 03] this idea has been employed in the context of DB&IR integration. For the XML fragments search from [Carmel 03] enclosing tags have been prepended to indexed words. In [Schenkel 07], the Wikipedia corpus has been enriched with XML tags. None of these systems uses any specialized index data structures, which does not let them scale well to large collections.

There are several works concerned with intuitive user interfaces for semantic search engines. Prominent examples are Haystack [Karger 05], Magnet [Sinha 05], and the Simile tools [Huynh 05]. Our proactive user interface is essentially that of our "normal" CompleteSearch engine and is inspired by the faceted-search paradigm [Hearst 02]. To our knowledge, our system is the first to combine semantic search with an interactive and proactive user interface.

9.4 The Query Engine

The CompleteSearch engine with its HYB data structure is at the core of our system. However, the query engine is modular and only requires the following characteristics:

- **Prefix search:** Given a *posting list, sorted by its document ids D*, and *a range W of word ids*, it must be able to compute the (sorted) posting list of all occurrences with a document id from the given set *D* and a word id from the given range *W*.
- **Join:** Given two posting lists, it must be able to compute the single posting list consisting of all items whose word ids occur in both lists, and which is sorted by document id.

By "posting list" we mean a list of tuples, where each tuple consists of (doc id, word id, position, score). The reason that we need to have both document ids and the corresponding word ids bundled is that we will make use of a kind of duality, e.g., outputting word ids while intersecting document ids or vice versa. Positional information is required as in Section 9.5, where we will carefully construct artificial documents, we will make heavy use of the positional information to put certain keywords at fixed positions. Scores for individual word-in-document pairs (or rather word-at-position pairs) are optional but will in practice always be present, as they allow for a highly customizable ranking of the results. HYB stores such sets as parallel lists, sorted by document id.

The first item in the list above (prefix search) essentially refers to the central Definition 1 and at the corresponding algorithmic problem, which HYB and AutoTree address. It is needed as in Section 9.5 we will use a mechanism to put database-like information into artificial documents similar to the one in Section 8.3. With this prefix search operation alone, we can already answer basic semantic queries of the following type. Assume that in our collection we have replaced each reference to *John Lennon* by the artificial word `musician:john_lennon`, and accordingly for all mentionings of a musician. We can then find all mentionings of a musician on pages mentioning the `beatles` by two prefix search queries: First, get the sorted list of all ids of documents containing the word `beatles`, by solving the prefix search query where *W* contains only the id of that word, and *D* is the set of all documents. Then perform another prefix+completion search where *D* is the list of these document ids and *W* is the list of ids of all words starting with `musician:`. This will give us the list of all mentionings of a musician in documents that also contain the word `beatles`. We will write the corresponding query as `beatles musician:*`. For another example, assume that every musician has its own document (as is the case in Wikipedia) and that, along with the artificial word for the musician's name, we also added the birth year as, for example, `borninyear:1940`. In the same manner as for the previous example, we would then obtain the list of all musicians born in 1940 by the query `borninyear:1940 musician:*`.

The join mechanism is needed, as our system depends on the ability to process db-style queries similar to those discussed in the Section 8.4. For example, consider the two prefix search queries from above: the first gave us a list of all musicians occurring in the context of the Beatles; the second gave us the list of all musicians born in 1940. Since the ids in both lists are of words of the same kind (artificial words starting with `musician:`), a join of these lists gives us the list of all musicians who are mentioned in the context of the Beatles *and* who were born in 1940. The list of document ids of the result list can be seen as "witnesses" of its individual items. Note that this example query *assembled information from different documents*; this is a kind of functionality which an ordinary inverted index cannot provide.

9.5 Mapping the Ontology to Artificial Words

We assume the ontology to be given as a directed graph, where the nodes are labeled with entities, and the arcs are labeled with names of relations. As a minimum, we require the relations *is a* and *subclass of* with the obvious (standard) semantics that will become clear by the following examples. For our application of the system to the Wikipedia, we picked the YAGO ontology from [Suchanek 07], which beyond the required *is a* and *subclass of*, contains relations such as *born in year*, *died in year*, and *located in*. YAGO was obtained by a clever combination of Wikipedia's category informations with the WordNet hierarchy [Fellbaum 98]. The YAGO graph has about 2.5 million arcs. Example arcs, written as ordered entity-relation-entity triples are *John Lennon is a musician*, *John Lennon born in year 1940*, *musician subclass of artist*.

In Section 8.3, we showed how arbitrary relational tables could be put into a form such that the CompleteSearch engine can process it. In the following, we describe how we cast YAGO, and similarly any other ontology which has at least the *is a* and *subclass of* relation, into *artificial words*, so that we can answer complex semantic queries efficiently using the two basic operations (prefix search and join) described in the previous section.

Ontology items as artificial words We assume that for each entity in the ontology there is a canonical document. For the Wikipedia collection and the YAGO ontology this is indeed the case; if it is not, we can simply add such canonical documents to the corpus. The construction that follows has, as a parameter, a set of *top-level categories*. The right setting of this parameter will be key to an efficient query processing. Intuitively, this set contains classes that are high up in the *subclass of* hierarchy, like *entity*, *person*, *substance*, etc.

Now consider an arc (x, r, y) from the ontology where x and y are the entities of the source and target node, respectively, and r is the relation of the arc. We then add the following artificial words to the canonical document for the entity x: At position 0, we add <c>:<x>, for each top-level category c of which x is an instance; at position 1, we add <r>:<p>, and at position p we add entity:<y>, where p is unique for relation r. For the special *is a* relation we further add, for each chain of triples $(x, is\ a, y_1)$, $(y_1, subclass\ of, y_2)$, ..., $(y_l, subclass\ of, z)$, the artificial word class:<z>.

For the three example triples from above, assuming that John Lennon is in the top-level categories entity and person, this would add (the first column gives the positions):

```
0   entity:john_lennon
0   person:john_lennon
1   is_a:2
2   class:musician
2   class:artist
1   born_in_year:3
3   entity:1940
```

Note that entity:john_lennon and person:john_ lennon are added only once, irrespectively of in how many triples the entity occurs, that all relations are added at position 1, and that the relation name contains a reference to the position of the entities from the target domain of the relation. Also note that there is no problem, if in the occurrence lists processed by the CompleteSearch engine several words occupy the same position in the same document.

Ontology queries Let us give a simple example for how we can make use of these artificial words. Assume we want to know the birth date of John Lennon. First, the query

```
entity:john_lennon + born_in_year:*
```

which is of the kind we have already discussed above, would give us the id of the canonical document for the entity John Lennon, as well as the (word id of the artificial word containing the) position 3. Then the query

```
entity:john_lennon + born_in_year:* ++ entity:*
```

gives us the (id of the word containing the) desired year. Here the pluses are CompleteSearch's proximity operators: <x> + <y> means that <y> must occur at the position following <x>, and n pluses say that the words must have a gap of $n-1$ positions between them. Analogously, the CompleteSearch engine provides the negative proximity operator -, with <x> - <y> meaning <y> + <x>[1]. Note that the basic definition of prefix search given at the beginning of Section 9.4 can easily be extended to perform proximity search; see Chapter 4 for details on how to incorporate positional information in HYB, and Section 5.5 for details on the general use of proximity search.

In Section 9.7, we will see that with artificial words added as described, we can handle arbitrary SPARQL queries. In Section 9.8 we will see how the artificial words together with the prefix search operation enable us to free the user from having to know any special syntax or names of relation in a completely interactive and proactive way.

[1]The symbols + and - are only used for clarity here. The syntax used internally is less intuitive (.,. and .;,), but these queries will never be entered by the user directly, as they are constructed by the user interface. The - is actually used for the NOT operator (see Section 5.7).

9.6 Entity Recognition and Combined Queries

The example queries in the previous section are purely semantic in the sense that they are operating on the ontology alone. In this section, we show how we can combine full-text and ontology search in an integrative manner, providing a functionality that is more than the sum of the two components.

Entity recognition We add, at the position of each occurrence of a word or phrase in the text collection that refers to an entity *x* from the ontology, the artificial word `<c>:<x>` for each top-level category *c* of which *x* is an instance. For example, if we take the same top-level categories as in the example from the previous subsection, then wherever *John Lennon* is mentioned (either by his full name, parts of his name, or however), we would add the artificial words

```
entity:john_lennon
person:john_lennon
```

Here we see that the set of top-level categories must not be too large: otherwise, a large number of artificial words would be added for each occurrence of an entity in the corpus, which would blow up our index beyond manageability.

Combined full-text and ontology queries Let us give an example of which kinds of queries are possible now. Assume that we want to find all occurrences of *persons* in documents that also contain the word `beatles` (see Section 9.9.2 for a discussion of when and why this kind of query makes sense). Then the simple prefix query `beatles person:*` would give us the desired list. But now assume that we are looking for all occurrences of *musicians* in the set of documents matching `beatles`. Further assume that musician is not a top-level category so that we do not have artificial words of the kind `musician:<x>`. However, note that in the canonical document of each entity *x* that is a musician, we have the artificial word `class:musician`. Then the query `class:musician - is_a:* - person:*` will give us a list of all (ids of words containing the names of) *persons* that are musicians, where - is the above-mentioned negative proximity operator. A simple join of this list with the list of the previous query will now give us the desired list of all *musicians* that occur in documents which also contain the word `beatles`.

In Section 9.7, we show that in this fashion any basic SPARQL graph-pattern query can be processed by a combination of prefix search and join operations.

Efficiency We have already seen that, in order to keep the index size small, we have to keep the number of top-level categories small, so that for each reference to an entity in the corpus, we add only few artificial words (one for each top-level category to which that entity belongs). The question arises, why we then not just take the top category *entity* (to which each entity belongs) as the only top-level category.

The problem is, that for a query like the one above, we would then have to execute the two queries

```
beatles entity:*
class:musician - is_a:* - entity:*
```

and join them. Now `entity:*` will have very many completions; indeed, one for every occurrence of *any* entity in the corpus. However, the prefix search queries can be processed efficiently only when the number of occurrences of words from the input word range (occurrences of words starting with `entity:` in this case) is not too large; see Chapter 4 for details. We must therefore choose the set of top-level categories such that for every sensible[2] query of the kind above, there is a top-level category above the category we are looking for, which does not have too many completions.

Realization for Wikipedia For our Wikipedia application, we identify occurrences of words or phrases that refer to an entity from the collection as follows. Recall our assumption that, without loss of generality, each entity in the ontology has a canonical document in the collection. Now Wikipedia has a lot of internal links, which, for selected words or phrases do exactly what we are looking for: they associate them with an entity. We use these links as training set for a simple but effective learning algorithm, which essentially follows the approach from [Dill 03].

In a nutshell, the approach proceeds in two phases: a training phase, and a disambiguation phase. In the training phase, we compute, for each word or phrase that is linked at least 10 times to a particular entity, what

[2] A query for all *entities* (people, substance, abstractions, etc.) associated with The Beatles will always be very expensive, for the reasons just explained. But it is not a very sensible query precisely because it is looking for entities from a very, very general class.

we call a *prototype vector*: this is a tf.idf-weighted, normalized list of all terms which occur in one of the neighborhoods (we consider 10 words to the left and right) of the respective links. Note that one and the same word or phrase can have several such prototype vectors, one for each entity linked from some occurrence of that word or phrase in the collection.

In the second phase, we iterate over all words or phrases that have been encountered in the training phase. For each of them, we compute the similarity to each of the possible prototype vectors, by adding up those values in the prototype vector which occur in the neighborhood of the word or phrase we are disambiguating. We then assign the meaning with the highest similarity. Similarly as in [Dill 03], we achieve a precision of around 90%, see Section 9.9.

9.7 SPARQL Queries

SPARQL [W3C 05] has been proposed by the W3C as a query language for ontologies. A SPARQL query corresponding to our (purely semantic) query "musicians born in 1940" from Section 9.4 would be:

```
SELECT ?who WHERE {
  ?who is_a musician .
  ?who born_in_year 1940 .
}
```

These so-called *basic graph patterns* are at the core of SPARQL, and for the purpose of this section we will consider them as instances of the following *binary constraint satisfaction problem (BCSP)* [Kumar 92]:

Given a directed graph G, a finite set S, for each node x a subset $S_x \subset S$ of values, and for each edge e of the graph an arbitrary relation $R_e \subseteq S \times S$. Then compute all possible assignments of values of S to the nodes of G that satisfy all relations, that is, all assignments such that for each edge e with value x assigned to its source node, and value y assigned to its target node, $(x, y) \in R_e$.

The example SPARQL query from above would correspond to a graph with three nodes x, y, and z, and two edges (x, y) and (x, z), where S_x is the set of all possible values, S_y consists of the single entity *musician*, S_z consists of the single entity *1940*, $S_{(x,y)}$ is the *is a* relation, and $S_{(x,z)}$ is the *born in year* relation.

The simplest algorithm for solving an instance of BCSP will iteratively *relax* the arcs of the given graph as follows: for an arc (x, r, y), where the current set of values for x is X, and the current set of values for y is Y, replace X by all values which are related, with respect to r, to a value from Y, and, analogously, replace Y by all values which are related, with respect to r, to a value from X. Like this, relax the nodes in some fixed order and repeat until the sets of values do not change anymore.

There are a number of more sophisticated algorithms making use of the same basic *relax* operation [Kumar 92]. It is not hard to see that in the important special case, where the query is a tree (and the vast majority of meaningful SPARQL queries are trees), it suffices to relax each arc exactly once (going from the leafs to the root). It is also not hard to see that, for our system, each relax operation is a matter of at most two prefix search queries and two join queries. The X and Y from above correspond to lists <X> and <Y> of occurrences in the CompleteSearch engine. To compute the set of all $y \in Y$ which are related, via r, to some $x \in X$, we first execute the query <X> + <r> ? <c>:* where c is the (top-level category encompassing the) domain of r, and ? is to be replaced by the proximity operator pertaining to r. A join of the matching completions for the prefix <c>:* in the query above with the result list of <Y> then gives the desired subset of Y. The desired subset of X is obtained analogously. We therefore have the following theorem.

Theorem: Following the approach above, we can process an arbitrary given basic SPARQL graph-pattern query with at most $2m$ prefix search and $2m$ join queries, where m is the number of relaxations required for the solution of the corresponding binary constraints satisfaction problem. If the SPARQL query is a tree, one relaxation per arc of that tree is sufficient.

9.8 User Interface

By "user interface" we do not mean the choice of colors and the like but really the interface, or "query translation device" between the user and the CompleteSearch engine in the background. Neither SPARQL nor

CompleteSearch's low-level query language (combinations of prefix searches and joins) are suitable for a front end to a search engine, where users are accustomed to extremely simple interfaces (namely, keyword search).

Inspired by the works of [Hearst 02] and addressing similar issues as for the case of faceted search (see Chapter 6), we have therefore devised an interactive and proactive user interface, which handles the most common types of semantic queries in a simple and intuitive manner. We have seen a first example in Figure 9.1. Here, we describe the other features in words and by example. Each of the following kinds of queries are a matter of a small SPARQL query that is a tree, and can therefore be solved efficiently by the theorem proved in the previous section.

Semantic completion of the last prefix This is the feature, an example of which is shown in Figure 9.1. At every keystroke, the system checks whether the last prefix of the current query string matches a class name. This is easily realized via artificial words as follows. For each class name, such as `musician` , we add an artificial word of the type `cn:musician:person` , where "person" would generally be the nearest top-level category containing the respective class, to a special document containing only such words. From the matching completions to the query `cn:music` , which is launched automatically in the background, we then get the information that the prefix `music` could also be interpreted as a class name. If more than one class name matches, a box with possible choices is displayed. For example, for the query `beatles music`, we get

```
Musical instrument (Object)
Music Genre (Relation)
Musician (Person)
```

Clicking on one of these choices then gives a picture similar to the one of Figure 9.1.

Proactive display of properties of an entity If we click on one of the musician's names in the lower box shown in Figure 9.1, the right panel will show documents referring to that musician prominently. On the left side, the lower box then shows a list of prominent properties of that entity according to the ontology, for example

```
John Lennon born in year 1940
John Lennon died in year 1980
John Lennon is a pacifist
```

Narrowing down a class Another frequent query is for entities from a given class which have a particular property. For example, assume we are looking for songs by German musicians. This can be formulated as the query `musician[german] song` for which the system displays occurrences of songs in documents that mention a musician, which in this or any other document occurs together with the word `german`. Note the subtle difference to the query `german musician song` for which all occurrences of songs in documents which contain the word German and mention a musician are displayed. Both kinds of queries are needed from time to time, but the user interface can be configured to translate the latter kind of query into the former automatically.

9.9 Experiments

We have implemented the whole system as described above. For the query engine we used the CompleteSearch engine, but the only required features are described in Section 9.4. The entity recognizer was described in Section 9.6, and the user interface in Section 9.8. We have applied it to the Wikipedia corpus combined with the YAGO ontology. Our version of the Wikipedia corpus has 2,863,234 documents and a raw size of the corresponding xml file of 8.0 GB. Our version of the YAGO ontology graph has 984,361 nodes (entities) and 2,505,638 arcs (facts). The total number of word occurrences, including the artificial words, is 1,513,596,408, which the CompleteSearch engine manages to hold in an index of size 4.1 GB, including 300 MB for the (compressed) vocabulary. Our experiments were run on a machine with 16 GB of main memory, 2 dual-core AMD Opteron 2.8 GHz processors (but we used only one core at a time), operating in 32 bit mode, running Linux. The ontology-only queries were run on an Intel Pentium 4, 3 GHz, with 1 GB of main memory, running Windows. We verified that the running times on these two machines are comparable.

9.9.1 Efficiency

As we discussed in Sections 9.2 and 9.3, efficiency aspects have hardly been considered for other semantic search engines with similar capabilities as the ones offered by our system. Preliminary experiments (not further reported in this chapter) have pointed to performance differences of up to two orders of magnitude.

For a more challenging performance assessment, we therefore devised the following, somewhat extreme, stress test. We constructed five query sets, three of which are purely ontological, while two others combine full-text and ontology search. For the pure ontology queries, preliminary experiments with Jena's ARQ [Seaborne 05], a state of the art SPARQL engine, have again pointed to performance differences of up to two orders of magnitude; we instead chose to compete with the highly tuned (ontology-only) search engine that comes with YAGO [Suchanek 07]. For the combined queries, we compare our system to a state of the art engine for full-text search; for this we took our own implementation which we already used as a baseline in Chapter 4.

This comparison is extremely unfair because both the ontology-only and the full-text only system are highly tuned toward their specific task, and cannot be used for the respective other task. Moreover, YAGO's ontology-only search is realized via a database management system with one large facts table, with indexes built over each possible attribute, which means that it has the set of answers for all basic fact queries precomputed. The space requirement is accordingly high: roughly 3 GB. This has to be compared with the about 4 GB which the CompleteSearch engine requires for the whole full-text + ontology index. A version of our system built for ontology search alone has an index size of only about 100 MB.

Queries We considered the following five queries sets. Note that queries from the first three sets can be answered from the ontology alone, and do not need the full-text search capability. For our experiments, we always used the complete index though.

Simple ontology queries: These ask for a list of triples from the ontology. Namely, for 1000 persons from the ontology, we ask for their birth year.

Advanced ontology queries: These queries require following paths in the ontology graph. Namely, for 100 relatively general classes, like biologists, social scientists, etc., we ask for all entities from that class.

Hard ontology queries: These queries require the combination of several facts from the ontology. Namely, for 1000 persons, we asked for the death dates of all persons that were born in the same year as the given person.

Combined full-text + ontology queries, Easy: These queries require the combination of full-text and ontology search as described in Section 9.5. For the easy set, we asked 50 queries for all counties of a given US state. The counties class is in our frontier set, so these queries can be processed via prefix search queries alone, e.g., `alabama counties:*`.

Combined full-text + ontology queries, Hard: These 50 queries ask for all computer scientists of a given nationality. Since the computer scientist class is not in our frontier set, but only the more general scientists class, these queries require more expensive prefix search queries as well as a join; see Section 9.6.

Results Table 9.1 summarizes the results of our efficiency experiments. Given the unfairness of the comparison, discussed above, the fact that the CompleteSearch's query processing times are comparable to the respective specialized baselines is a strong result in favor of our approach.

For the full-text only queries, we simply replaced all entity prefixes by corresponding index words, e.g., `alabama county` or `german computer scientist`. Note that these queries hardly retrieve any relevant documents. We provide these figures merely to show that unlike the systems discussed in Section 9.3, CompleteSearch manages to stay in the same order of magnitude as state of the art full-text search. Also note that for the full-text only search, the hard query set can be processed faster than the easy set because there are more occurrences of the word `county` than of the word `scientist` in the Wikipedia. Finally, note that the CompleteSearch engine manages to keep also worst-case (max.) query processing times low (which was in fact one main design goal as discussed in Chapter 4); this is especially important for the interactive, suggest-as-you-type user interface.

9.9.2 Search Result Quality

The *quality* of the search results provided by our system, or any other semantic search engine of a similar kind, depends on three main factors: (1) the quality of the ontology; (2) the quality of the entity recognizer; and (3)

	CompleteSearch		Onto-only or Full-text only	
	avg.	max.	avg.	max.
Onto Simple	2 ms	5 ms	3 ms	20 ms
Onto Advanced	9 ms	31 ms	3 ms	794 ms
Onto Hard	64 ms	208 ms	78 ms	550 ms
Onto + Text Easy	224 ms	772 ms	90 ms	498 ms
Onto + Text Hard	279 ms	502 ms	44 ms	85 ms

Table 9.1: Query processing times on five query sets

the principal ability of the combined full-text and ontology search to provide interesting results.

Quality of the ontology We employ an existing ontology, namely YAGO from [Suchanek 07]. In that paper the authors estimate, by extrapolation from human assessment on a sample, that 95% of YAGO's facts are correct.

Quality of the entity recognizer Table 9.2 shows the quality of our entity recognizer, described in Section 9.6. For this assessment we held out 10% of the words or phrases for which the corresponding entities are known (because they link to some Wikipedia page), and measured the percentage of entities recognized correctly (precision). We compare our implementation (OUR) against two simple baselines: the naive scheme that assigns every word to the most common sense that has been encountered for that word in the training set (TOP), and the scheme that assigns every word to a random sense (RANDOM).

Scheme	all words	2 senses	3 senses	≥ 4 senses
OUR	93.4%	88.2%	84.4%	80.3%
TOP	91.9%	83.5%	77.2%	77.6%
RANDOM	71.5%	50.2%	33.4%	14.0%

Table 9.2: The precision of the entity recognizer used by our system

Combined ontology and full-text queries quality Since there are no benchmarks for combined full-text and ontology queries on the Wikipedia, we came up with the following two *generic* (as opposed to hand-crafted) query sets:

People associated with universities (PEOPLE, 100 queries): We took the first 100 lists from the Wikipedia page "Category:Lists of people by university in the United States", for example "List of Carnegie Mellon University people". For each such list, we generated a combined ontology and full-text query, for example, `carnegie + mellon + university person:*` and computed the percentage of relevant entities which appear in the result list (RECALL) and the percentage of relevant entities among the top 10 (P@10), considering the entities listed on the respective Wikipedia page as the relevant ones.

Interestingly, our system found a number of false-negatives: for example, among the top ten entities for the above query, Andrew Carnegie was returned, who is not listed on the respective Wikipedia page.

Counties of American states (COUNTIES, 50 queries): For the second query set, we took 50 Wikipedia lists of the form "List of counties in <US state>".

Results As shown in Table 9.3, CompleteSearch enriched with the ontological knowledge achieves an almost perfect recall and a reasonable precision on both query sets. Unfortunately we could achieve these precisions only *without* the additional entities learned by our recognizer, while recall is not affected much. This may sound

paradoxical at first but there is a simple explanation: the amount of information provided by the Wikipedia links (our training data) is already very complete, e.g., at least the first mentioning of a person on a Wikipedia page almost always contains a link to that person's entry in the Wikipedia. This means that for the broad kinds of queries from our two sets, tagging more entities only creates more noise. The entity recognizer would obviously help if we had additional documents with interesting information, but without human-labeled entities, for example, news articles. Here we could then, e.g., ask for politicians mentioned in a news article containing the word "nuclear power", or for countries occurring together with "tax increase". This issue requires further investigation.

	PEOPLE	COUNTIES
P@10	37.3%	66.5%
RECALL	89.7%	97.8%

Table 9.3: Precision and recall on two query sets

Chapter 10

User Study

10.1 Introduction

Fast response time is one of the keys to user satisfaction: for user interfaces in general, and for knowledge management systems in particular. Our CompleteSearch engine was designed to provide a set of "intelligent" search features, yet with response times suitable for a fully interactive (per-keystroke) user interface, even for very large amounts of text.

In this chapter, we describe how we combined CompleteSearch with our institute's helpdesk system, which contains over 7,000 records from a few hundred users. We conducted a small user study with five members from our helpdesk staff, who each performed ten typical tasks, alternatingly using CompleteSearch and Google Desktop Search (`desktop.google.com`).

All five users preferred CompleteSearch over Google Desktop, mainly because of its speed, the feeling of being in power, and the enhanced search facilities. The interactive behavior with its instant response time was unanimously perceived as the single, greatest strength of the system. Although this did not come unexpected for us, it is somewhat surprising given the many other intelligent (semantic) search features provided by CompleteSearch, and given that Google Desktop Search is by no means slow (though not as interactive) either.

In Section 10.3, we describe the adaptations we made to the CompleteSearch engine to optimize it for the helpdesk setting. In Section 10.4, we discuss work related to the problem of experience management in general, and to helpdesk systems in general. In Section 10.5, we describe the setup of our user study and discuss the results.

Following our study, the helpdesk staff is now using CompleteSearch on a daily basis. For privacy reasons, we cannot offer a demo of the CompleteSearch setup for our helpdesk system. However, a list of related demos, can be found at `http://search.mpi-inf.mpg.de/`.

10.2 The Helpdesk System

Our institute's helpdesk system is essentially a database of *short text fragments* grouped into threads, called *tasks*. Each fragment has an author, user privileges, etc. and each task has a priority, a status, someone responsible for it, etc. Customers can initiate or react to a task by email or via a web interface. At the time of our user study, the database contained about 7,300 tasks. Tasks are in the literature sometimes also referred to as 'trouble tickets'.

10.3 Adapting the CompleteSearch Engine to the Helpdesk System

The screenshot of Figure 10.1 shows our CompleteSearch engine in action for the collection of helpdesk tasks.

Note that the basic autocompletion search feature (see Section 5.2) is a purely syntactic one: it is all about words starting with a particular sequence of letters. We next show, by a few examples, how by adding appropriate artificial words to the index, a surprising variety of more and more semantic features can be realized, too. The examples here are for the particular use case of our helpdesk system. A far more thorough study of

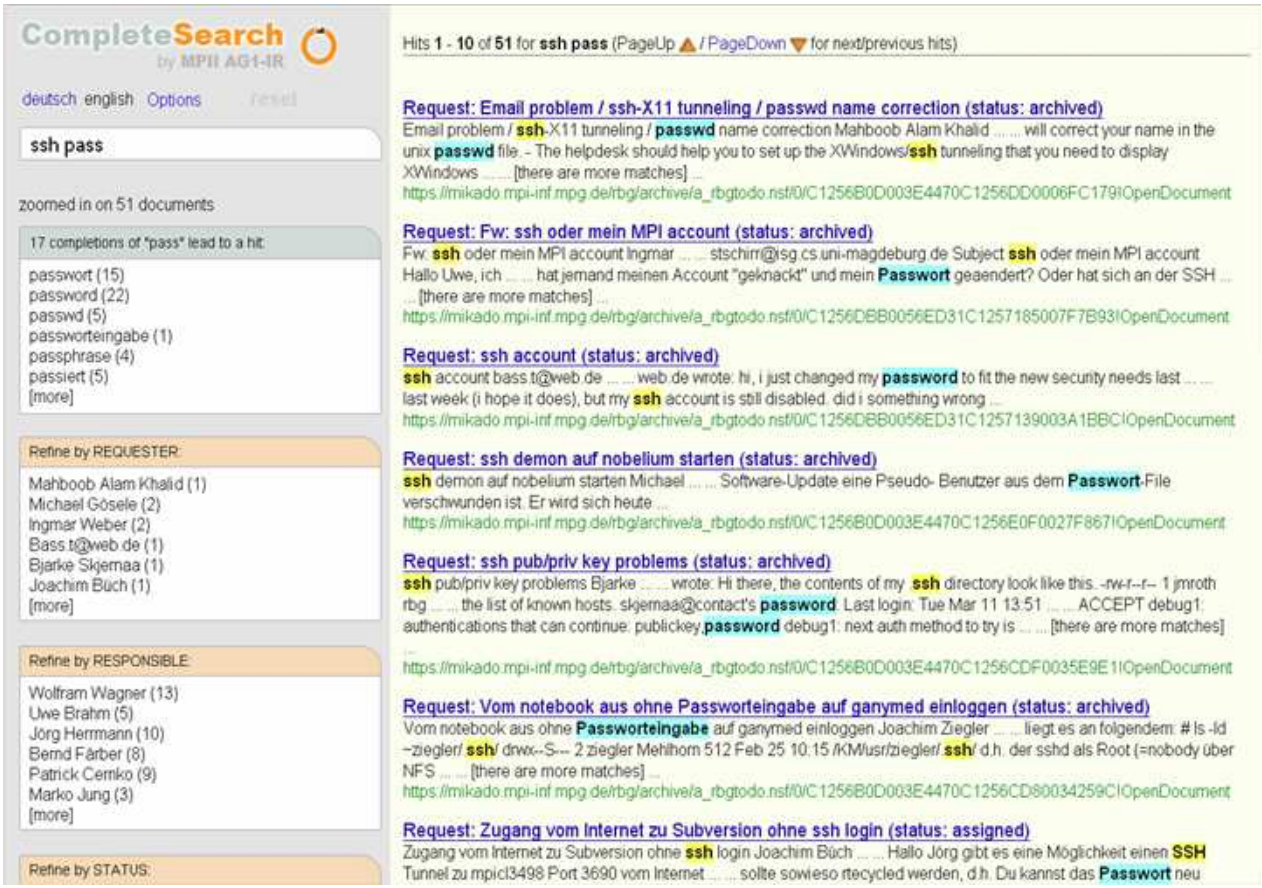

Figure 10.1: A screenshot of the CompleteSearch engine in action for the query `ssh pass` on a collection of over 7,000 tasks of our helpdesk system, indexed with full text + category information. The list to the right shows documents which contain a word starting with `ssh` together with a word starting with `pass`. The first box below the search field shows words, subwords, and phrases starting with `pass` (passwort, password, passphrase, etc.) that lead to the best hits. The other boxes show a breakdown of the whole set of 2,302 hits by various categories: requesting user, who is responsible, task status, etc. All this information is updated instantly after every single keystroke, hence the absence of any kind of search button. All features are obtained via one and the same highly efficient and scalable prefix search and completion mechanism.

how the CompleteSearch engine can be used to allow more semantic queries was presented in the previous Chapter 9.

Semantic annotations. Assume the word `kmhp81001` has been identified, as a printer name. We would then add the words `printer:kmhp81001` and `kmhp81001:printer` to the index. If a user then typed `problem pri`, one of the completions might be `printer:kmhp81001` (which in our current GUI would be displayed as `kmhp81001, the PRINTER`), provided that there is at least one document in the collection which contains both `problem` and `kmhp81001`. And similarly so for the query `problem kmhp`. For our institute, we could simple obtain an exhaustive inventory list of all printers, notebooks and desktop, for which we then added this information.

Faceted search. Assume that a document is known to belong to certain categories, say `linux` and `mail`. We would then add the words `cat:linux` and `cat:mail` to the system. Then, for the query `network slow cat:`, the list of completions would give a breakdown into categories of the hits for the query `network slow`. In the actual GUI, users do not have to type the `cat:` but a query with this word appended is launched automatically after every keystroke. See Figure 10.1 for an example. This feature, known as *faceted search* and described in detail in Chapter 6, gives the user complete freedom to alternate between keyword-based searching and directory-style browsing. This category information, e.g., the current status of the request or the person responsible, was already provided with the document itself in the database in the background.

More features could be added in this vein, following the approach given in Section 8.3. With the help of DB-style joins (see Section 8.5), we could then even find information spread over several documents, for

example: which PhD students have notebooks of a particular type. Here one document could mention the name of the student (without mentioning his position) and the type of his notebook, and another document could mention that this particular person is PhD student. Other features of the CompleteSearch engine, such as proximity search (see Section 5.5) or the OR and NOT operators (see Section 5.7), work out-of-the-box.

10.4 Related Work

In our user study, we concentrated on the *retrieval* aspect of managing a helpdesk system. Indeed, in a study of Brandt [Brandt 02], only 14% of all helpdesk calls were new problems that required serious attention. For the helpdesk system of our institute, where the majority of the customers are computer experts themselves, this percentage is somewhat larger, but finding existing bits of information is still the predominant task. Most of the existing helpdesk systems can be viewed as a combination of some kind of content management system with some kind of search engine [Sinnett 04].

In case-base reasoning (CBR), equal attention is paid to the problem of *adding* new cases to the system (in order to maximize utility for the solution of similar future cases) [Leake 96; Bergmann 04]. While we did not directly address this problem in our user study, the combination of CompleteSearch with any kind of content management system shares many desirable features with full-blown CBR systems like HOMER [Göker 99]. Such CBR systems require a hierarchy of category labels that is carefully adjusted to the application at hand. These category labels are then used to guide the search for existing cases similar to a given one, as well as the task of storing new cases in the right place in the hierarchy. By appropriately adding these category labels to the search index, CompleteSearch, with its context-sensitive completion facility, has the potential for achieving much of this too. We consider an in-depth investigation of this relationship as a very promising direction for future research.

10.5 The User Study

10.5.1 Setup of Study

For our study we asked five volunteers from the helpdesk staff to use both Google Desktop Search (GDS) and our CompleteSearch (CS) system to each process a set of ten fictitious problems, modeled after typical helpdesk requests. For example, the first two problems were:

P1. I'll be away for a couple of weeks soon: How can I configure my IMAP account to send an automatic reply "I'll be back in X weeks"?

P2. When I run matlab on the compute server 'dude' some windows with dialogues (e.g., to save/open files) have all the text unreadable, i.e., only symbols rather than letters are displayed. What can I do?

After a minimum of 1 minute and a maximum of 5 minutes, each participant answered the following nine multiple choice questions for each problem:

M1. How much time did you spend on this request? [1-5 minutes]

M2. How realistic was the given problem? [1 = completely artificial/would never happen, 4 = could happen/has happened]

M3. How easy was it to find enough relevant information for this request? [1 = impossible, 4 = trivial]

M4. How many relevant documents did you find? [None, One or two, three to five, six to ten, more than 10]

M5. How empowered by the system did you feel? [1 = I felt powerless/very lost, 4 = I had all the control and power one could hope for]

M6. GDS only: Did you use GDS advanced search syntax? [yes, no]

M7. CS only: Did you use the categories (status, requester, responsible)? [yes, no]

M8. CS only: Did you use the proximity ('..') or phrase ('.') operator? [yes, no]

M9. CS only: Did you use the completions under the search box? [yes, no]

After having attempted to find information for all ten of the problems, they were given six more questions concerning the whole CS system and its perceived benefits. For example, the first two questions were:

G1. Overall, in your everyday work, would you use GDS or CS? [Always GDS, mostly GDS, mostly CS, always CS]

G2. For the particular helpdesk application, distribute a total of 20 points among the following eight features. Design, speed, useful completions, useful categories, proximity and phrase search, high quality ranking, high quality snippets, usability

Finally, there was also a short personal interview at the end to have a chance to get more explicit feedback.

10.5.2 Division of Problem Sets Into Two Halves

The set of 10 problems was split into two halves. Each of the five participants used GDS for one half and CS for the other. Three participants first used CS and then GDS, two used the systems in the other order. In total, both CS and GDS were used for 25 problems. This way we tried to ensure that (a) all problems were attempted by both systems (to avoid biases from differences between easy and difficult problems) and (b) all participants used both systems (to avoid biases from differences between lenient and critical users).

As it turned out, the first half of the problems was significantly more difficult. The average ease of finding on a 4-point scale (question M3) was 2.2 for the first half set compared to 2.6 for the second half. Correspondingly, the number of relevant documents found (question M4) was smaller for this set. The average selected category (of five possible) was only 2.4 compared to 3.2 for the other set. Our system happened to be used more often on the difficult set of queries.

10.5.3 Main Findings of Study

All users prefer CS. Four of the five test users always prefer CS (question G1). One user mostly prefers CS. [1]

It's easier to find information with CS. All except one test user found it easier to find results with CS than with GDS. For these four users the average response to Question M3 (ease of finding information) was higher when using CS than when using GDS.[1] The overall average ease of finding, on a 4-point scale with more meaning "easier" was 2.6 for CS vs. 2.2 for GDS.

More relevant documents are found with CS. Correspondingly, more relevant documents were found with CompleteSearch (question M4). The average selected category, on a 5-point scale with 1 being no relevant document at all, was 3.0 for CS vs. 2.6 for GDS.

Users feel more empowered by CS. Four of the five test users felt more empowered by CS than by GDS. For these participants the average response to Question M5 was higher for CS than for GDS.[1] The overall average "feeling of empowerment" was 3.0 for CSE vs 2.3 for GDS.

Problems were found to be realistic. To ensure the relevance of the study to the helpdesk's daily business, we asked the participants to evaluate the realism of the given problems. The average rating was 3.4 on a 4-point scale, with 1 corresponding to "completely artificial/would never happen" and 4 corresponding to "could have/has happened before".

Relative importance of features. Question G2 was targeted at the relative importance of various ingredients of a search engine as perceived by the user. The users had to distribute a total of 20 points among eight features, which means that a score above 2.5 indicates an important feature.

The three most important ones turned out to be (i) general speed (3.3), (ii) proximity and phrase queries (3.0) and (iii) to present good completions to the user (2.8). Indeed, proximity and phrase queries were tried for 19 of the 25 problems for which CS was used. Users stated to have used the displayed completions (in whatever way) for 13 of the same set of 25. Interestingly, the GDS advanced syntax (phrase queries, negation operator) was used for only 7 of 25 problems.

Observations from interview. All participants unanimously explained that, for their everyday use, the categories (see Section 10.2) *are* actually useful, despite the fact that in the study they only used this information to refine search results for two of 25 problems. The reason for this is that the category information is *not* very useful for information finding tasks (where any category can contain relevant information), but *is* useful for navigational tasks (where the student is looking for a *particular* request and might remember the current status of that request).

All users stated that they heavily depended on the short document previews to select documents to inspect more closely. One user asserted that with CS he never (!) went to the actual document (which takes about 2-5 seconds to open), but simply used CS's feature to increase the size of the snippet. This way he could view all of the document's content in less than a second, without leaving the search interface. Interestingly, in the list of 8 most important aspects (question G2), high quality snippets only got an average score of 2.3, leaving it 6th in the list.

Two of the three users who used CS for the first set of queries unsolicitedly mentioned that they felt a considerable "oh no"-effect, when they had to use GDS for the second half. Now they had to type more, had to wait longer for their results and had no option to adjust the size of the snippets.

[1] The single 'outlier' used CS for the more difficult first problem set.

Some of the participants of our study pointed out that a lot of their search time is devoted to repeated query formulations in both English and German. As a partial solution, we have, after the study, integrated ‘semantic tags’ for certain words. For example, all terms starting with any of the prefixes ‘druck’, ‘print’ or ‘lpr’ (and others) are now tagged to belong to the category ‘printer’.

Chapter 11

Implementation and Design Choices

11.1 Introduction

Without an efficient data structure, such as HYB (see Chapter 4 or AutoTree (see Chapter 3) to handle autocompletion search queries (according to Definition 1), our CompleteSearch engine would not nearly be as powerful as it is. Still, when building the system, there were many crucial decisions to be made, which were not necessarily of an algorithmic nature but often more of a software engineering nature. They all had a direct impact on the usability (as perceived by the end user), the extensibility (with respect to new features) or the efficiency.

11.2 Maximizing Locality of Access

It is a truism that sequential access to data is faster than random access. For a typical disk, average seek time is 5 milliseconds versus an average transfer rate of 50 Megabytes per second. But even when the data is entirely in main memory, sequential access is up to 100 times faster than random access. This factor tends to be smaller for complex applications (or programs in higher-level languages, see the paragraph after the next), but when other factors of inefficiency are eliminated it plays a crucial role. Indeed, our first index data structure AutoTree is theoretically close to optimal, in that we could prove its query processing time to be asymptotically bounded by the size of the output, for corpora with a realistic average case behavior. Yet, our follow-up scheme HYB without this theoretically desirable property, but highly optimized for locality of access, beats AutoTree by a factor of 1.2 – 4.0, depending on the input/output volume of the query. See Section 4.6 for details.

11.3 Minimizing the Amount of Data to Read

To reduce the amount of data that have to be read from disk and processed per query, the HYB index makes extensive use of compression. Further, it is one of the distinguishing features of HYB that the index data is laid out such that the processing of a query requires mere *scans* of portions of the data. In particular, no sorting or other non-linear or non-local operations of large portions of the data are required to find the matching word-in-document pairs. Such operations are only performed on the (small) set of these matching pairs (for scoring) when they are in main memory, but not on the whole data that is processed from disk to obtain them.

11.4 Choosing the Right Programming Language

Concerning implementation of the core compute module, C++ was the programming language of choice.[1] It is often debated how much faster an implementation in C++ really is compared to, say, a program written in JAVA, or queries to a DBMS like Oracle or MySQL. Indeed, anecdotal evidence as well as a study by Prechelt [Prechelt 00] have it that the choice of programming language does not make much of a difference for the average program. However, when it comes to algorithms highly optimized for sequential access to data, the

[1] Of course, the user interface used AJAX (Php and Javascript) and in preprocessing steps we extensively used Perl.

C++	JAVA	MySQL	Perl
1800 MB/s	300 MB/s	16 MB/s	2 MB/s

Figure 11.1: Average processing rate (in Megabytes per second) for four different programming languages/environments scanning an ordinary (in-memory) array of 10 million 4-byte integers, measured on a Linux PC with two 3 GHz Intel Xeon processors and 4 GB of main memory. The rate for C++ is close to the 2 GB/s memory bandwidth specified for that machine.

difference is enormous. See Table 11.1, where for a simple scanning task, C++ wins over JAVA by factor of 6, over a MySQL application by a factor of more than 100, and over a scripting language like Perl by a factor of almost 1000.

We made extensive use of *templating*, to reduce the code complexity without compromising instruction-cache efficiency (few instructions in the inner loops) and branch predictability (no conditionals in the inner loops, wherever possible).

11.5 The Right Building Blocks

HYB's and also AutoTree's inherent duality, in the sense that they always work with word-in-document pairs, made the later addition of the join functionality (see Section 8.5) easy. Rather than intersecting the document ids, we now had to intersect the word ids. The intersection algorithm suddenly is different (as word ids are not sorted), but apart from that the same framework holds. Similarly, it allowed us to easily integrate customizable scoring for the matching completions, as we also had the scores for each word-in-document pair.

These parallel lists of document ids, word ids, positions and scores, are also the main building blocks to be processed by our system. All central routines, such as intersection or join, take as input two such lists and output a new one. This choice of data encapsulation helped to facilitate the addition of new features.

11.6 Keeping the Core System Simple

It proved to be of enormous benefit to have chosen just the right level of abstraction for the central mechanism. The prefix search and completion mechanism, with the join functionality added later, turned out to be powerful enough to provide a wide range of features, yet simple enough to allow for an efficient implementation. The addition of these advanced features later, which were not foreseen at all when we started with our system, turned out to generally require no or only very minor modifications of the core completer engine. Most features such as faceted search (Chapter 6) or even the whole semantic search (Chapter 9) could be implemented by (i) choosing the right artificial words to add and (ii) modifying the user interface slightly.

11.7 Hiding the Complexity From the User

We tried to design the system such that the user has to know as little as possible about the features available and their syntax. In fact, he could even choose to ignore the basic prefix search mechanism and simply type full words as usual. Then our system would work as a normal search engine. Similarly, in cases where categorical information is available, the user does not need to know about the syntax of the artificial words to exploit the faceted search feature. This becomes even more relevant for the semantic search feature, which is built on top of such an intricate use of artificial words that no user could be expected to handle it directly. In all of these cases the mechanism to hide the complexity from the user is the same: the user interface launches complex queries automatically in the background and, if they give the appropriate results, displays this additional information to the user, who can ignore this information completely or try to make use of it.

11.8 Result Caching

Besides the use of an efficient data structure such as HYB to process autocompletion search queries, the single most important "detail" to obtain short processing times, is a clever reuse of results already computed in the (recent) past. Recent queries (and the corresponding results) are kept by our system in a cache, which we refer to as "history". All the use cases discussed below are independent of the algorithm used to compute the results, and apply equally well to INV, AutoTree or HYB.

The simplest application of the history would be to do a simple lookup for the exact same query and, if found, return the cached result. This mechanism is already required to process a normal sequence of autocompletion search queries, corresponding to a user typing one query word after the other. Once the user starts typing a new (partial) word, we first do a lookup for the part of the query preceding this last prefix. If it is in the history, as it always will be if the query has been typed normally, we then immediately obtain the result for this first part, including the D (according to Definition 1) to be used in combination with the last prefix. Without this mechanism, a query with k prefixes/words would be roughly a factor k slower.

The mechanism, which we call "result filtering", was already mentioned in Section 2.1. We dicuss it here again for completeness. It has a less dramatic effect than the crucial result caching above, but it is still of very practical relevance. It is applicable to query sequences such as `info`, `infor`, ..., `information`, or `information ret`, ..., `information retrieval`. Here, we can exploit the fact that the sequence of autocompletion search queries is of the form (D, W), (D, W'), (D, W''), ..., where $W \supseteq W' \supseteq W'' \supseteq \ldots$. The result of the current/last query, say Φ'', is hence a subset of the result Φ' for the previous query. That is, we merely have to scan the elements $(d, w) \in \Phi'$ once and filter out those for which $w \in W''$. This is in practice always faster than launching a full recomputation.

A slightly more intricate method of filtering works, when the W in a sequence such as above remains constant but the D gets smaller (where both W and D are with respect to the very last prefix). This can happen when, as described in Section 6.3.1, we always add the same constant prefix after each query typed by the user. For the case of faceted query, we would, e.g., always add the prefix `cat:author` to get a breakdown of the results by author. While typing `retrieval`, the following sequence might be generated in this way: `retr cat:author`, `retri cat:author`, ..., `retrieval cat:author`. To optimize the performance for this case, we do the following. We get the list for the last prefix the user has entered, say `retriev`, and intersect it with the result list for the last query from the sequence in the history, say `retrie cat:author`. This way we avoid processing the potentially long list for `cat:author` in such a setting altogether. In a similar fashion, we can obtain the result for `information ret cat:author` by intersecting the results for `information ret` and `information cat:author`.

11.9 Running Several Threads in Parallel

Although our HYB data structure was built with the aim of minimizing worst case running times in mind, there are still instances where our CompleteSearch engine cannot guarantee a fully interactive response time. This happens when the query is of a very unspecific nature such as `pro` or `the`[2], where the input volume to process is enormous and the output volume is of a similar size. (Also see the following section.) During times, when the query load is very high, a single worst case query could cause many other request to wait for an unacceptable amount of time. This can be avoided by processing each request in a separate CPU thread.

Making the CompleteSearch engine multi-threaded, was relatively straight-forward, as most (small) data elements, such as counters, did not need to be shared between threads. The only data element, which *must* be shared for efficient processing, is the history, with its cached results of recent queries. Here problems could occur if two threads both try to concurrently create the same entry in the result cache, or if one thread tries to read a recent result, which is still under construction. In such cases, the second thread needs to wait for the first one to finish, before it can access the cached result.

[2] We usually do not remove any stop words, such as "the" or "and", from the documents.

11.10 Not Making Life Harder Than Necessary

The hypothetical query `t`, which asks for all documents containg a word starting with "t", ranked by relevance, along with a list of such words, would for the Wikipedia collection take more than a minute for our CompleteSearch engine. But this, and similar queries, are of no practical use to the user, and so we simply disallow them. That is, we require a certain minimal length of a prefix, while it is typed, before we (i) launch any query at all and (ii) do a full prefix search, rather than treating it as an exact match query (without the implicit "*" at the end). This takes a significant amount of load of the system, without sacrificing the usability of the system. In practice, we choose a minimum length of 2 – 3, before we launch a query. For this minimum length, the last prefix is interpreted as a full word and a "*" is *not* yet implicitly added. For prefixes of at least 3 – 4 letters, we then provide the usual autocompletion search feature.

11.11 Putting Everything Together

Building an interactive web application like CompleteSearch that is supposed to display its GUI via any standard web browser, was a very challenging task.

It starts with the design, which is all but obvious. The completion server necessarily has a non-negligible start-up cost and cannot be started from scratch for every query, but has to run as a background process continuously. But letting the client's web browser communicate with a program on a remote computer is a security problem. We solved this by a three-step approach: the web page displayed to the client contains JavaScript code, which for each user action triggers the loading of a special web page via an AJAX[3] protocol. This web page is dynamically created via PHP, in particular taking care of the communication with the completion server, and generating the HTML as well as the JavaScript code. Figure 11.2 shows a simple diagram of the data flow.

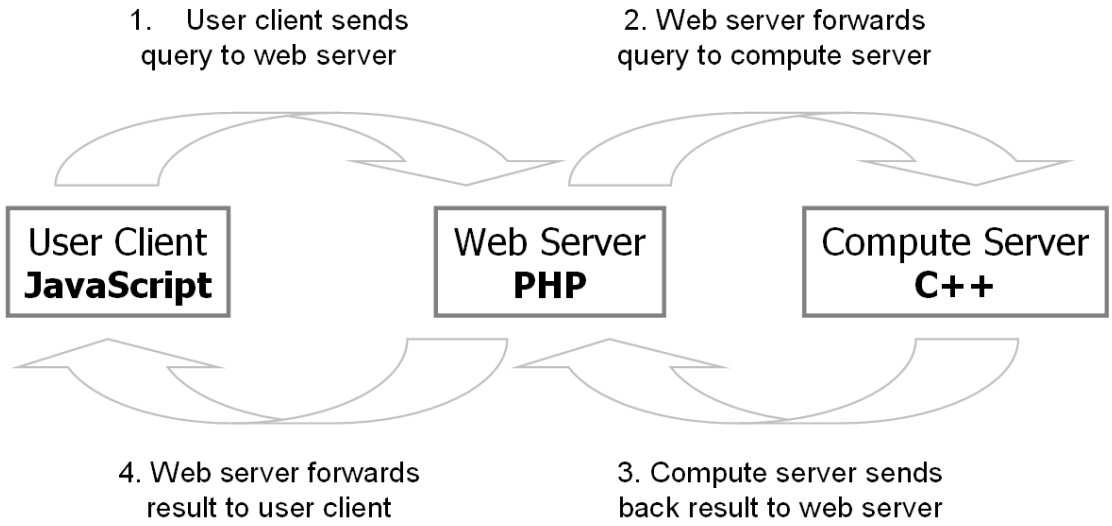

Figure 11.2: The communication between the user client and the web server is done via AJAX. The web server exchanges data directly with the compute server via socket communication.

The advantage of this approach is that no installation or special software is required on the side of the user; any standard web browser will do. Nor can any firewall settings be a problem: if web browsing works, CompleteSearch works too. The price for this is complex code on three different machines (completion server, web server, client machine) which interacts with each other in a non-trivial manner, and can be hard to debug. Missing standards and inconsistencies concerning the way web browsers process JavaScript, render a complex layout, or deal with the browser history (back button) are a constant source of trouble.

[3] "Asynchronous JavaScript and XML". Uses the JavaScript XMLHttpRequest object to trade data with the web server, without reloading the page.

11.12 Keeping in Touch with the Users

The CompleteSearch engine would not be close to what it is today without the feedback of our users. In this section we report on some of the main lessons we learned from this feedback loop.

The first users were ourselves. When starting the project 3 years ago, we first wrote a prototype (in Perl) to see the search engine in action, on a real collection. Many of the features were born in that way, e.g., the search within tags (see Section 5.6), such as the “From” or the “Subject” field in a collection of email messages from a newsgroup.

One of the lessons we had to learn was that the vast majority of (our) users is not willing to read even the tiniest bit of documentation before using a search engine, not even if the search does not give the expected results. Actually, we anticipated this to some extent, and tried to keep the user interface intuitive and simple right from the beginning. And after all, the whole approach of CompleteSearch is a proactive one: display completions, hits, refinements, alternatives, etc. as the user is typing. If he or she opts to ignore this information, the basic functionality of a search engine is still there.

But the following surprised us: below the search field we put a very short note saying ”Type ? for help”, and the mechanism was such that typing ? at any point in the query would instantaneously display a few sentences on the most important advanced operators which can help improve search results. Well, hardly any user ever pressed the ? key, let alone read the help information. After this experience we abandoned all our plans for more elaborate help pages, feedback forms, etc. and focused on making our whole system as proactive as possible.

Still, we have not given up on receiving feedback at least on a smaller scale. Our user study (see Chapter 10) was an opportunity to do this. The personal, short interviews with the volunteers at the end gave us valuable insights into the “user perspective”, e.g., clarifying how they use which kind of information on a regular basis. We also continue to receive feedback from interested colleagues, who use our system to search in a collection of scientific articles. See `http://search.mpi-inf.mpg.de/` for a list of available online demos.

Chapter 12

Conclusions

12.1 Recap of Main Contributions

We have built a search engine, called CompleteSearch, which efficiently supports a wide range of features. It is built on top of a mechanism to solve *prefix search and completion* queries: Given a set of documents (containing all the query words typed so far) and a prefix (corresponding to the query word currently being typed), find (i) the most promising completions of the prefix which, if fully typed, would yield at least one matching document from the given set, as well as (ii) the most relevant such documents.

In the first part of this dissertation, we formalized this problem (Chapter 2) and presented two data structures, AutoTree (Chapter 3) and HYB (Chapter 4), which address the corresponding algorithmic problem. AutoTree has the desirable property that, for realistic corpora and queries, its running time depends linearly on the size of the output. HYB is optimized for scenarios, where the index is too large to fit in main memory, and offers space bounds in terms of the empirical entropy of the corpus. For both data structures, we performed extensive experiments to evaluate their performance relative to a baseline using the inverted index, and also to evaluate their performance relative to each other. In a general setting, HYB outperforms AutoTree.

In the following chapters, we then discussed various extensions and applications of our CompleteSearch engine. These included a list of features, such as ranking or proximity search, which are applicable to any search engine, and which do not rely on the prefix search and completion mechanism for an efficient realization (Chapter 5), but also more advanced features, which depend on this mechanism, such as faceted search of hierarchical information (Chapter 6), interactive completion to synonyms (Chapter 7), support for database-style queries (Chapter 8) and efficient search of ontologies (Chapter 9). We also combined our CompleteSearch engine with the database of the helpdesk of our institute. This we used to perform a small user study, to prove the usefulness of your system (Chapter 10). All of the additional features are provided in a proactive manner. That is, the user does need to know about the feature, and can even ignore it completely. Some of the lessons learned from building our system, and some crucial details not directly related to the the other chapters, were discussed in Chapter 11.

We obtained an efficient realization of these advanced features mentioned above using one and the same mechanism, namely our context-sensitive prefix search and completion, by only adding suitable words to the documents. Note the relevance to a web scenario, where users have control over their documents, but not over the search engine.[1]

12.2 Loose Ends and Possible Improvements

Although our CompleteSearch engine has certainly matured well beyond the stage of a mere ephemeral prototype to be only used to give running times for a comparative experiment, there are still a number of issues, which can be improved. These issues are discussed in the following sections.

[1] It will indeed be interesting to see, if any major search engine will in the near future offer (small) content-providers the possibility, to provide additional semantic information about the pages, or parts thereof, in a well-defined format, e.g., by using special key words/tags for persons or place names, going well beyond the html <meta> tags. Such information could be harvested to improve the search quality and to offer new features, but it also poses the usual risk of creating just another source of search engine spam.

12.2.1 Improving the User Interface

The user interface could be improved in several ways. For example, categories with a hierarchical structure should be displayed in a tree-like fashion and not as a flat list, as we currently do. Furthermore, the user should in such a setting have the possibility to limit his search scope to any of the subcategories. Fortunately, as a hierarchical structure (e.g., 'Subject'->'Computer Science'->'Information Retrieval') can easily be encoded via the appropriate prefixes (e.g., `cat:subject:computer_science:information_retrieval`), this does not pose any principal problem.

Some users also pointed out the lacking possibility to display completions for one of the *previous prefixes*, when a user, after having typed several prefixes, moves the cursor back to edit an earlier term. Here, it might be desirable, to show completions for the prefix, which is currently being edited, rather than for the last prefix. This feature could be easily implemented by duplicating the prefix, currently focused on, and appending it (again) to the end of the query. This could be done automatically by the user interface and the completions returned would then naturally be for the prefix of interest.[2]

Another nice-to-have feature would be a button to remove query words that have been added by a refine-by-category operation. Similarly, it might be desirable to select more than one category at the same time (and then display matches from any of these). So far we concentrated on providing certain new features, and showing that we can support them very efficiently. Over time, and with helpful input from our users, these additional improved features will be added.

12.2.2 Improving the Algorithms

As, ultimately, the user is only interested in the *most relevant* documents (and completions), our approach of first computing *all* matches, and then rank them in a second phase, seems to leave room for improvement in terms of efficiency. Indeed, standard techniques for *top-k retrieval* [Fagin 03; Bast 06a] seem to lend themselves nicely to this problem. They address the issue of efficiently reporting the top k items from multiple lists, which pertain to the best aggregated score (where, e.g., scores from different lists are summed for a fixed item). All approaches try to limit the depth to which the lists involved have to be scanned, hoping that the best k items can be reported without scanning all the lists to the end, which is exactly what one would hope to achieve for the HYB algorithm.

Unfortunately, none of the approaches is immediately applicable without non-trivial modifications. The reason for this is that, while it is sufficient to only *display* the, say, top 10 document from the set D' to the user, we need the *full* such set, if the user starts typing a new prefix, as the D' now becomes the new D. Two possible approaches to remedy this would be as follows. (i) We could give up on the strict left-to-right order of processing an autocompletion search query. Then the "usual machinery" would apply directly, as we are dealing with a (small) number of lists which need to be (partially) intersected according to document ids, while reporting elements with high aggregated scores as early as possible. This, if implemented unmodified, would most likely incur higher processing costs, as we are no longer exploiting the fact that we have already computed the result for all parts, except the very last part of the query. (ii) We could implement a more involved re-use of previously computed items (see Section 11.8), such that no full recomputation of the new D is required. E.g., we could compute the top 100 documents, but only display the top 10 to the user. Then, ideally, in the computation for the following result, these 100 results will contain the new top 10 results. Obviously, as more and more prefixes are added to the query, we will have to go back and extend our knowledge, i.e., compute more hits, for previous entries in the history. In the same spirit, we could compute the top 10 hits quickly, and present these to the user, but at the same time, in the background, continue to compute *all* (or a sufficiently large number of) hits, so that for anticipated future requests the required information is already precomputed. This way, we would use the time the user spends inspecting the results for useful computations.

These issues are worth further investigation, especially as the corpus size (and the result size) continue to grow.

[2] There is a small caveat involved here, if this prefix is part of a phrase (`inform.ret`) or is required to be close to another prefix (`unit..ameri`). Then one needs to use the corresponding "inverse proximity operator", similar to the use of positive and negative proximity operators in Section 9.5.

12.2.3 Improving the Index Management

In Section 10.4, we mentioned the similarity between the classical case-based reasoning approach used in systems like Homer, and a corresponding tagging used in conjunction with the context-aware completion mechanism of CompleteSearch. It would indeed be interesting to integrate the CompleteSearch engine into a content management system, such that users can input the documents (and tags) themselves directly.

A closely related aspect, to which we have paid little attention so far, is the question of how to deal with *dynamic updates*. So far, our philosophy has been to split large collections into several parts, to have one small part into which all the changes are immediately incorporated, and to rebuild partial indices from scratch when it becomes necessary. In the IR world this is actually considered one of the most effective ways of updating [Lester 04]. Still, there is work to do for us here, especially in automating this process.

There is also the issue of distributing our indices over several machines, to be able to scale up to not just millions but billions of documents. Both standard techniques [Moffat 06], either using a term based partitioning (where each machine becomes responsible for a subset of key words) or a document based partitioning (where each machine becomes responsible for a subset of documents), immediately apply to our setting.

12.2.4 Evaluating the Search Quality

Finally, the general focus of this dissertation has been on the *efficient* realization of certain features. The question of the *quality* of the search results, or even the features themselves has generally been left open. The results from our user study (Chapter 10), however, already provided strong indications that the approach of proactively and efficiently providing advanced search features has a positive practical impact. It would be interesting to conduct a user study on the perceived usefulness of the individual features, in particular concerning the type of semantic queries supported by our system (Chapter 9), and, indeed, semantic queries in general.

As with all the features provided by the CompleteSearch engine, the hope is that they allow the user to find *more relevant information in less time* than with a traditional search engine, while requiring *less knowledge* about the corpus at hand.

Bibliography

[Aggarwal 88] Alok Aggarwal & Jeffrey S. Vitter. *The Input/output Complexity of Sorting and Related Problems*. Communications of the ACM, vol. 31, no. 9, pages 1116–1127, 1988.

[Alstrup 00] Stephen Alstrup, Gerth S. Brodal & Theis Rauhe. *New Data Structures for Orthogonal Range Searching*. In 41st Symposium on Foundations of Computer Science (FOCS'00), pages 198–207, 2000.

[Amer-Yahia 05] Sihem Amer-Yahia, Pat Case, Thomas Rölleke, Jayavel Shanmugasundaram & Gerhard Weikum. *Report on the DB/IR Panel at SIGMOD 2005*. SIGMOD Record, vol. 34, no. 4, pages 71–74, 2005.

[Anh 05] Vo Ngoc Anh & Alistair Moffat. *Inverted Index Compression Using Word-Aligned Binary Codes*. Information Retrieval, vol. 8, pages 151–166, 2005.

[Arentz 04] Will A. Arentz & Aleksander Øhrn. *Multidimensional Visualization and Navigation in Search Results*. Lecture Notes in Computer Science, vol. 3212, pages 620–629, 2004.

[Arge 99] Lars Arge, Vasilis Samoladas & Jeffrey S. Vitter. *On Two-dimensional Indexability and Optimal Range Search Indexing*. In 18th Symposium on Principles of Database Systems (PODS'99), pages 346–357, 1999.

[Baeza-Yates 04] Ricardo Baeza-Yates. *A Fast Set Intersection Algorithm for Sorted Sequences*. Lecture Notes in Computer Science, vol. 3109, pages 400–408, 2004.

[Bast 05] Holger Bast & Debapriyo Majumdar. *Why Spectral Retrieval Works*. In Gary Marchionini, Alistair Moffat, John Tait, Ricardo Baeza-Yates & Nivio Ziviani, editors, 28th Annual International Conference on Research and Development in Information Retrieval (SIGIR'05), pages 11–18, 2005.

[Bast 06a] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald & Gerhard Weikum. *IO-Top-k: Index-access Optimized Top-k Query Processing*. In 32nd International Conference on Very Large Data Bases (VLDB'06), pages 475–486, 2006.

[Bast 06b] Holger Bast, Christian W. Mortensen & Ingmar Weber. *Output-Sensitive Autocompletion Search*. In 13th Symposium on String Processing and Information Retrieval (SPIRE'06), volume 4209 of *Lecture Notes in Computer Science*, pages 150–162, 2006.

[Bast 06c] Holger Bast & Ingmar Weber. *Type Less, Find More: Fast Autocompletion Search with a Succinct Index*. In 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR'06), pages 364–371, 2006.

[Bast 06d] Holger Bast & Ingmar Weber. *When You're Lost for Words: Faceted Search with Autocompletion*. In SIGIR'06 Workshop on Faceted Search, pages 31–35, 2006.

[Bast 07a] Holger Bast, Alexandru Chitea, Fabian Suchanek & Ingmar Weber. *ESTER: Efficient Search on Text, Entities, and Relations*. In Charles Clarke, Norbert Fuhr & Noriko Kando, editors, 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR'07), pages 671–678, 2007.

[Bast 07b] Holger Bast, Debapriyo Majumdar & Ingmar Weber. *Efficient Interactive Query Expansion with CompleteSearch*. In 16th Conference on Information and Knowledge Management (CIKM'07), page to appear, 2007.

[Bast 07c] Holger Bast & Ingmar Weber. *The CompleteSearch Engine: Interactive, Efficient, and Towards IR&DB Integration.* In 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07), pages 88–95, 2007.

[Bast 07d] Holger Bast & Ingmar Weber. *Managing Helpdesk Tasks with CompleteSearch: A Case Study*. In Norbert Gronau, editor, 4th Conference on Professional Knowledge Management (WM'07), pages 101–108, 2007.

[Bergmann 04] Ralph Bergmann, Klaus-Dieter Althoff, Sean Breen, Mehmet Göker, Michel Manago, Ralph Traphöner & Stefan Wess. Developing industrial case-based reasoning applications. Springer-Verlag GmbH, 2nd edition, 2004.

[Bickel 05] Steffen Bickel, Peter Haider & Tobias Scheffer. *Learning to Complete Sentences*. In 16th European Conference on Machine Learning (ECML'05), pages 497–504, 2005.

[Billerbeck 05] Bodo Billerbeck. *Efficient Query Expansion*. PhD thesis, RMIT University, 2005.

[Binding 04] Ceri Binding & Douglas Tudhope. *KOS at Your Service: Programmatic Access to Knowledge Organisation Systems*. Journal of Digital Information, vol. 4, no. 4, 2004. http://rapid.isd.glam.ac.uk/FACET/.

[Boncz 06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger & Jens Teubner. *MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine*. In Conference on Management of Data (SIGMOD'06), pages 479–490, 2006.

[Brandt 02] D. Scott Brandt. *Techman's Techpage: Automating Your IT Help Desk*. Computers in Libraries, vol. 22, no. 3, pages 52–55, 2002.

[Brants 00] Thorsten Brants. *TnT – A Statistical Part-of-Speech Tagger.* In 6th Conference on Applied Natural Language Processing (ANLP'00), pages 224–231, 2000. http://www.coli.uni-saarland.de/~thorsten/tnt.

[Carmel 03] David Carmel, Yoëlle S. Maarek, Matan Mandelbrod, Yosi Mass & Aya Soffer. *Searching XML Documents Via XML Fragments*. In 26th Annual International Conference on Research and Development in Information Retrieval (SIGIR'03), pages 151–158, 2003.

[Castells 07] Pablo Castells, Miriam Fernndez & David Vallet. *An Adaptation of the Vector-Space Model for Ontology-Based Information Retrieval*. IEEE Transactions on Knowledge and Data Enginering, vol. 19, no. 02, pages 261–272, 2007.

[Chaudhuri 00] Surajit Chaudhuri & Gerhard Weikum. *Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System*. In 26th International Conference on Very Large Data Bases (VLDB'00), pages 1–10, 2000.

[Chaudhuri 05] Surajit Chaudhuri, Raghu Ramakrishnan & Gerhard Weikum. *Integrating DB and IR Technologies: What is the Sound of One Hand Clapping?* In 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05), pages 1–12, 2005.

[Chazelle 88] Bernard Chazelle. *A Functional Approach to Data Structures and its Use in Multidimensional Searching*. SIAM Journal on Computing, vol. 17, no. 3, pages 427–462, 1988.

[Clarke 05] Charles L. A. Clarke, Nick Craswell & Ian Soboroff. *The TREC Terabyte Retrieval Track*. SIGIR Forum, vol. 39, no. 1, page 25, 2005.

[Darragh 90] John J. Darragh, Ian H. Witten & Mark L. James. *The Reactive Keyboard: A Predictive Typing Aid*. IEEE Computer, vol. 23, no. 11, pages 41–49, 1990.

[Deerwester 90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas & Richard A. Harshman. *Indexing by Latent Semantic Analysis*. JASIS, vol. 41, no. 6, pages 391–407, 1990.

[Demaine 00] Erik D. Demaine, Alejandro Lopez-Ortiz & J. Ian Munro. *Adaptive Set Intersections, Unions, and Differences*. In 11th Symposium on Discrete Algorithms (SODA'00), pages 743–752, 2000.

[Dill 03] Stephen Dill, Nadav Eiron, David Gibson, Daniel Gruhl, Ramanathan V. Guha, Anant Jhingran, Tapas Kanungo, Kevin S. McCurley, Sridhar Rajagopalan, Andrew Tomkins, John A. Tomlin & Jason Y. Zien. *A Case for Automated Large-scale Semantic Annotation*. Journal of Web Semantics, vol. 1, no. 1, pages 115–132, 2003.

[English 02] Jennifer English, Marti Hearst, Rashmi Sinha, Kirsten Swearingen & Ka-Ping Yee. *Hierarchical Faceted Metadata in Site Search Interfaces*. In Conference on Human Factors in Computing Systems (CHI'02), pages 628–639, 2002.

[Fagin 03] Ronald Fagin, Amnon Lotem & Moni Naor. *Optimal Aggregation Algorithms for Middleware*. Journal of Computer and System Sciences, vol. 66, no. 4, pages 614–656, 2003.

[Fellbaum 98] C. Fellbaum, editor. Wordnet: An electronic lexical database. MIT Press, 1998.

[Ferragina 03] Paolo Ferragina, Nick Koudas, S. Muthukrishnan & Divesh Srivastava. *Two-dimensional Substring Indexing*. Journal of Computer and System Science, vol. 66, no. 4, pages 763–774, 2003.

[Ferragina 05] Paolo Ferragina & Giovanni Manzini. *Indexing Compressed Text*. Journal of the ACM, vol. 52, no. 4, pages 552–581, 2005.

[Ferragina 06] Paolo Ferragina, Raffaele Giancarlo & Giovanni Manzini. *The Myriad Virtues of Wavelet Trees*. In 33rd International Colloquium on Automata, Languages and Programming (ICALP'06), pages 560–571, 2006.

[Finkelstein 01] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman & Eytan Ruppin. *Placing Search in Context: The Concept Revisited*. In 10th International World Wide Web Conference (WWW10), pages 406–414, 2001.

[Fonseca 05] Bruno M. Fonseca, Paulo Braz Golgher, Bruno Pôssas, Berthier A. Ribeiro-Neto & Nivio Ziviani. *Concept-based Interactive Query Expansion*. In 14th Conference on Information and Knowledge Management (CIKM'05), pages 696–703, 2005.

[Fuhr 98] Norbert Fuhr & Thomas Rölleke. *HySpirit — A Probabilistic Inference Engine for Hypermedia Retrieval in Large Databases*. In 6th International Conference on Extending Database Technology (EDBT'98), LNCS 1377, pages 24–38, 1998.

[Gaede 98] Volker Gaede & Oliver Günther. *Multidimensional Access Methods*. ACM Computing Surveys, vol. 30, no. 2, pages 170–231, 1998.

[Göker 99] Mehmet Göker & Thomas Roth-Berghofer. *The Development and Utilization of the Case-Based Help-Desk Support System HOMER*. Engineering Applications of Artificial Intelligence, vol. 12, pages 665–680, 1999.

[Grabski 04] Korinna Grabski & Tobias Scheffer. *Sentence Completion*. In 27th Annual International Conference on Research and Development in Information Retrieval (SIGIR'04), pages 433–439, 2004.

[Grossi 00] Roberto Grossi & Jeffrey S. Vitter. *Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching*. In 32nd Symposium on the Theory of Computing (STOC'00), pages 397–406, 2000.

[Grossi 03] Roberto Grossi, Ankur Gupta & Jeffrey S. Vitter. *High-Order Entropy-Compressed Text Indexes*. In 14th Symposium on Discrete Algorithms (SODA'03), pages 841–850, 2003.

[Grossi 04] Roberto Grossi, Ankur Gupta & Jeffrey S. Vitter. *When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications*. In 15th Symposium on Discrete algorithms (SODA'04), pages 636–645, 2004.

[Hearst 02] Marti Hearst, Ame Elliott, Jennifer English, Rashmi Sinha, Kirsten Swearingen & Ka-Ping Yee. *Finding the Flow in Web Site Search*. Communications of the ACM, vol. 45, no. 9, pages 42–49, 2002.

[Hearst 06] Marti A. Hearst. *Clustering Versus Faceted Categories for Information Exploration*. Communications of the ACM, vol. 49, no. 4, pages 59–61, 2006.

[Heinz 03] Steffen Heinz & Justin Zobel. *Efficient Single-pass Index Construction for Text Databases*. Journal of the American Society for Information Science and Technology, vol. 54, no. 8, pages 713–729, 2003.

[Huynh 05] David Huynh, Stefano Mazzocchi & David R. Karger. *Piggy Bank: Experience the Semantic Web Inside Your Web Browser*. In 4th International Semantic Web Conference (ISWC'05), pages 413–430, 2005.

[Jakobsson 86] Matti Jakobsson. *Autocompletion in Full Text Transaction Entry: A Method for Humanized Input*. In Conference on Human Factors in Computing Systems (CHI'86), pages 327–323, 1986.

[Kabra 03] Navin Kabra, Raghu Ramakrishnan & Vuk Ercegovac. *The QUIQ Engine: A Hybrid IR DB System*. In 19th International Conference on Data Engineering (ICDE'03), pages 741–743, 2003.

[Karger 05] David R. Karger, Karun Bakshi, David Huynh, Dennis Quan & Vineet Sinha. *Haystack: A General-Purpose Information Management Tool for End Users Based on Semistructured Data*. In 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05), pages 13–26, 2005.

[Klein-Heyl 07] Christian Klein-Heyl. *Improving Search Engines: Finding Infixes and Phrases*, 2007. Master thesis under the supervision of Holger Bast.

[Kumar 92] Vipin Kumar. *Algorithms for Constraint-Satisfaction Problems: A Survey*. AI Magazine, vol. 13, no. 1, pages 32–44, 1992.

[Leake 96] David B. Leake. Case-based reasoning: Experiences, lessons, and future directions. AAAI Press/MIT Press, 1996.

[Lester 04] Nicholas Lester, Justin Zobel & Hugh E. Williams. *In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems*. In 27th Australasian Computer Science Conference (ACSC'04), 2004.

[MacKay 02] David J. C. MacKay. Information theory, inference & learning algorithms. Cambridge University Press, 2002.

[Mäkinen 04] Veli Mäkinen & Gonzalo Navarro. *New Search Algorithms and Space/time Tradeoffs for Succinct Suffix Arrays*. Rapport technique C-2004-20, University of Helsinki, 2004.

[Manber 90] Udi Manber & Gene Myers. *Suffix arrays: a new method for on-line string searches*. In 1st Symposium on Discrete algorithms (SODA'90), pages 319–327, 1990.

[McCreight 85] Edward M. McCreight. *Priority Search Trees*. SIAM Journal on Computing, vol. 14, no. 2, pages 257–276, 1985.

[Metzler 04] Donald Metzler, Trevor Strohman, Howard Turtle & W. Bruce Croft. *Indri at TREC 2004: Terabyte Track*. In 13th Text Retrieval Conference (TREC'04). National Institute of Standards & Technology, 2004.

[Mishra 92] Priti Mishra & Margaret H. Eich. *Join Processing in Relational Databases*. ACM Computing Survey, vol. 24, no. 1, pages 63–113, 1992.

[Moffat 96] Alistair Moffat & Justin Zobel. *Self-Indexing Inverted Files for Fast Text Retrieval*. ACM Transactions on Information Systems, vol. 14, no. 4, pages 349–379, 1996.

[Moffat 06] Alistair Moffat, William Webber & Justin Zobel. *Load balancing for term-distributed parallel retrieval*. In 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR'06), pages 348–355, 2006.

[Munro 96] J. Ian Munro. *Tables*. In 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96), pages 37–42, 1996.

[Paynter 00] Gordon W. Paynter, Ian H. Witten, Sally J. Cunningham & George Buchanan. *Scalable browsing for large collections: A case study*. In 5th Conference on Digital Libraries (DL'00), pages 215–223, 2000.

[Porter 80] Martin F. Porter. *An Algorithm for Suffix Stripping*. Program, vol. 14, no. 3, pages 130–137, 1980.

[Prechelt 00] Lutz Prechelt. *An empirical comparison of seven programming languages*. IEEE Computer, vol. 33, no. 10, pages 23–29, 2000.

[Puglisi 06] Simon J. Puglisi, William F. Smyth & Andrew Turpin. *Inverted Files Versus Suffix Arrays for Locating Patterns in Primary Memory.* In 13th Symposium on String Processing and Information Retrieval (SPIRE'06), pages 122–133, 2006.

[Raghavan 01] Sriram Raghavan & Hector Garcia-Molina. *Integrating Diverse Information Management Systems: A Brief Survey*. IEEE Data Engineering Bulletin, vol. 24, no. 4, pages 44–52, 2001.

[Rønn-Jensen 06] Jesper Rønn-Jensen. *Live Search Explained*, 2006. http://justaddwater.dk/2006/01/26/live-search-explained/.

[Ross 05] Kenneth A. Ross & Angel Janevski. *Querying Faceted Databases*. Lecture Notes in Computer Science, vol. 3372, pages 199–218, 2005.

[Schenkel 07] Ralf Schenkel, Fabian M. Suchanek & Gjergji Kasneci. *YAWN: A Semantically Annotated Wikipedia XML Corpus*. In 12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW'07), 2007.

[Seaborne 05] Andy Seaborne. *ARQ — A SPARQL Processor for Jena*, 2005. http://jena.sourceforge.net/ARQ.

[Shneiderman 00] Ben Shneiderman, David Feldman, Anne Rose & Xavier F. Grau. *Visualizing Digital Library Search Results with Categorical and Hierarchical Axes*. In 5th Conference on Digital Libraries (DL'00), pages 57–66, 2000.

[Sinha 05] Vineet Sinha & David R. Karger. *Magnet: Supporting Navigation in Semistructured Data Environments*. In Conference on Management of Data (SIGMOD'05), pages 97–106, 2005.

[Sinnett 04] Christian J. Sinnett & Tammy Barr. *OSU Helpdesk: A Cost-effective Helpdesk Solution for Everyone*. In 32nd Annual Conference on User Services (SIGUCCS'04), pages 209–216, 2004.

[Spink 02] Amanda Spink, Bernard J. Jansen, Dietmar Wolfram & Tefko Saracevic. *From E-Sex to E-Commerce: Web Search Changes*. IEEE Computer, vol. 35, no. 3, pages 107–109, 2002.

[Steinbrunn 97] Michael Steinbrunn, Guido Moerkotte & Alfons Kemper. *Heuristic and Randomized Optimization for the Join Ordering Problem*. International Journal on Very Large Data Bases, vol. 6, no. 3, pages 191–208, 1997.

[Stocky 04] Tom Stocky, Alexander Faaborg & Henry Lieberman. *A Commonsense Approach to Predictive Text Entry*. In Conference on Human Factors in Computing Systems (CHI'04), pages 1163–1166, 2004.

[Stonebraker 05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran & Stan Zdonik. *C-store: A Column-oriented DBMS*. In 31st International Conference on Very Large Data Bases (VLDB'05), pages 553–564, 2005.

[Suchanek 07] Fabian Suchanek, Gjergji Kasneci & Gerhard Weikum. *YAGO: A Core of Semantic Knowledge*. In 16th World Wide Web Conference (WWW'07), pages 697–706, 2007.

[Swami 89] Arun Swami. *Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques*. In Conference on Management of Data (SIGMOD '89), pages 367–376, 1989.

[Theobald 05a] Martin Theobald, Ralf Schenkel & Gerhard Weikum. *Efficient and Self-tuning Incremental Query Expansion for Top-k Query Processing*. In 28th Annual International Conference on Research and Development in Information Retrieval (SIGIR'05), pages 242–249, 2005.

[Theobald 05b] Martin Theobald, Ralf Schenkel & Gerhard Weikum. *An Efficient and Versatile Query Engine for TopX Search*. In 31st International Conference on Very Large Data Bases (VLDB'05), pages 625–636, 2005.

[Trotman 04] Andrew Trotman & Börkur Sigurbjörnsson. Narrowed extended XPath I (NEXI). http://www.cs.otago.ac.nz/postgrads/andrew/2004-4.pdf, 2004.

[van Dongen 00] Stijn van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000. http://micans.org/mcl.

[Völkel 06] Max Völkel, Markus Krötzsch, Denny Vrandecic, Heiko Haller & Rudi Studer. *Semantic Wikipedia*. In 15th World Wide Web Conference (WWW'06), pages 585–594, 2006.

[Voorhees 94] Ellen M. Voorhees. *Query Expansion using Lexical-Semantic Relations*. In 17th Conference on Research and Development in Information Retrieval (SIGIR'94), pages 171–180, 1994.

[Voorhees 04] Ellen Voorhees. *Overview of the TREC 2004 Robust Retrieval Track*. In 13th Text Retrieval Conference (TREC'04), 2004. http://trec.nist.gov/pubs/trec13/papers/ROBUST.OVERVIEW.pdf.

[W3C 05] W3C. *The SPARQL Query Language*, 2005. World Wide Web Consortium, http://www.w3.org/TR/rdf-sparql-query.

[Williams 99] Hugh E. Williams & Justin Zobel. *Compressing Integers for Fast File Access*. Computer Journal, vol. 42, no. 3, pages 193–201, 1999.

[Witten 99] Ian H. Witten, Timothy C. Bell & Alistair Moffat. Managing gigabytes: Compressing and indexing documents and images, 2nd edition. Morgan Kaufmann Publishers Inc, 1999.

[Yee 03] Ka-Ping Yee, Kirsten Swearingen, Kevin Li & Marti Hearst. *Faceted Metadata for Image Search and Browsing*. In Conference on Human Factors in Computing Systems (CHI'03), pages 401–408, 2003.

[Zobel 98] Justin Zobel, Alistair Moffat & Kotagiri Ramamohanarao. *Inverted Files Versus Signature Files for Text Indexing*. ACM Transactions on Database Systems, vol. 23, no. 4, pages 453–490, 1998.