Effiziente Text Suche

Christian Ortolf

3. August 2008

Zusammenfassung

In diesem Seminar Paper wird ein Überblick geboten über Suchstrukturen die verwendet werden können um die Suchdauer unabhängig der Textgröße zu gestalten. Insbesondere soll die Funktionsweise des Invertierten Index erklärt werden da dies die vorherrschende Suchstruktur für Textsuchen ist.

1 Einführung

Textsuche ist ein Problem das in einer Vielzahl von Applikationen vorkommt. Wir definieren uns das Problem als eine Suche über eine Menge von Dokumenten die alle Dokumente zurück liefern soll welche der Suchanfrage entsprechen. Im Gegensatz zu Suchen in Datenbanken sind Suchanfragen bei Textsuchen eher simpel. Anstelle von komplettem SQL werden nur 2 Arten von Anfragen benötigt.

- **Boolsche Anfragen** Einzelne Suchworte mit AND (Alle Suchbegriffe müssen vorhanden sein) oder OR (Ein Suchbegriff muss vorhanden sein) logisch verknüpft.
- **Phrase Querys** Anfrage auf Begriffe die alle vorhanden und in einer Reihenfolge vorkommen müssen.

Zudem wird Text der Indexiert wird normalisiert. Das heist das Problem der Textsuche wird weiter reduziert in dem man ein Dokument zu einer Liste aus Termen reduziert. Diese ergeben sich in dem man Sonderzeichen wie z.B. Freizeichen oder Schrägstriche als Worttrenner behandelt Jedes so erhaltene Wort gilt nun als Term der weiter normalisiert wird. So ist zum Beispiel bei der Suche die Groß und Kleinschreibung irrelevant und wird daher aus den Termen entfernt. Zudem können sprachspezifische Spezialzeichen auch umgeschrieben werde, z.B: ä zu ae oder ß zu ss. Des weiteren wird so genanntes stemming betrieben. Hierbei wird jeder Term auf seinen Wortstamm reduziert, da es für den Suchenden oft nicht ankommt ob ein Wort nun in der Vergangenheitsform vorliegt oder eine besondere Endung besitzt.

2 Suffix Baum

Ein Baum wird aufgebaut in dem jeder Suffix des Textes in den Baum eingefügt wird. Jeder Knoten Symbolisiert dabei ein Buchstaben. So kann man nun anhand des Suchbegriffs dem Baum entlangwandern. Da ein simples Einfügen der

Buchstaben jedoch zu einem Platzverbrauch von $O(n^2)$ führen würde wird statt dessen nur Referenzen auf Textstellen eingefügt. Somit kann man dann über diese Referenzen nicht nur Platz sparen sondern auch das Dokument finden zu dem gezeigt wird.

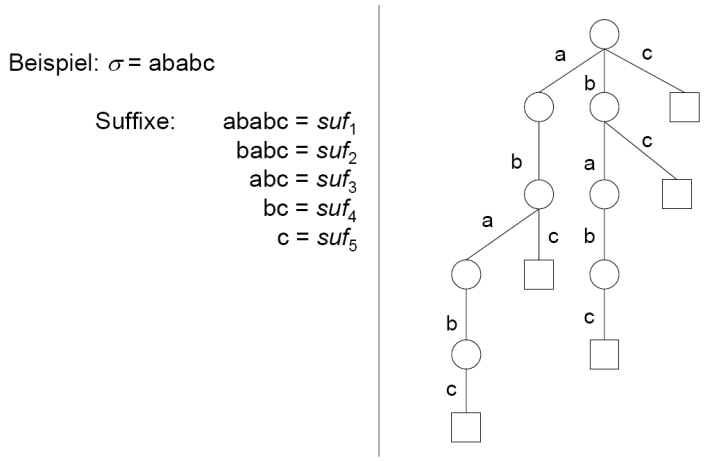

Abbildung 1: Suffix Tree Beispiel

3 Suffix Arrays

Suffix Arrays sind eine Datenstruktur welche den länger bekannten Suffix Bäume verbessern sollte. Da Suffix Bäume obwohl mit dem Speicherverbrauch in O(n) in der Praxis einen zu großen Speicherverbrauch haben. Die Literatur spricht hier von 24n-28n [1] als Speicherverbrauch. Dies kommt daher das jeder Knoten eine große Anzahl links auf Folgende Knoten Speichern muss. Das Suffix Array löst dies indem es aus einem einzigen Array besteht das Referenzen auf alle Suffixe des Textes hält. Diese werden Sortiert und können dann später schnell mit einer Binärsuche gefunden werden. Um mehrere Dokumente mit einem Suffix Array zu bearbeiten hängt man sie aneinander zu einer einzigen Zeichenkette. Um nun ein einzelnes Dokument zu finden Braucht man nun nur noch eine lookup Tabelle um Aus der Globalen Position wieder zu einem Dokument aufzulösen.

Die Größe des Suffix Arrays verglichen mit dem Text wird mit 9n benannt. Jedoch gibt es inzwischen Komprimierte Varianten bei denen von einer Größe von 0.7n-1.3n gesprochen wird. Dies jedoch selbst verständlich nur wenn es mit natürlich Sprachlichem Text benutzt wird. Suffix Arrays haben jedoch gerade im Genetik Bereich eine Anwendung um z.B. Subsequenzen eines Genstrangs zu finden oder wenn Reguläre ausdrücke über einem Text evaluiert werden müssen.

4 Inverted Index

Der Invertierte Index ist die Dominante Suchstruktur die von Suchmaschinen verwendet wird. Hierbei wird aus jedem Dokument eine Liste aus Termen extrahiert. Aus allen Termen aller Dokumente wird ein Wörterbuch aufgebaut. Jedes

Wort hat nun eine Invertierte Liste mit Dokumenten verweisen angehängt. So kann man schnell mit Hilfe eines Wortes die das Wort enthaltenden Dokumente finden.

Indem man nicht nur die Dokumenten IDs abspeichert sondern auch noch die Position des Terms im Dokument erhält man auch noch die Möglichkeit Phrase Querys zu beantworten in dem man prüft ob die gefragten Dokumenten IDs in aufsteigender Reihenfolge sind. Normale Boolsche Querys brauchen diese Information nicht. Sie liefern einfach eine Vereinigung (OR) oder einen Schnitt (AND) der invertierten Listen.

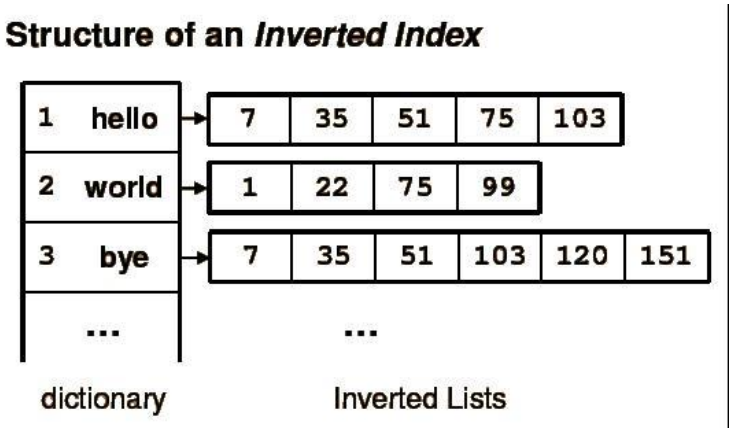

Abbildung 2: Invertierter Index Beispiel

Um den Speicherverbrauch des Invertierten Index zu senken muss man es schaffen diese Invertierten Listen möglichst klein abzuspeichern. Da sie nur aus IDs für Dokumente also aus Integern bestehen hat es sich bewährt hierfür verschiedene Integer Kompressionsverfahren zu verwenden.

4.1 Integer Kompression

Anstatt einfach nur niedrigere Bit Zahlen für feste Integer zu verwenden z.B. bei unter 100.000 Dokumenten würden 17 Bit ausreichen. Wenn man nun in den Listen nur Delta werte angibt (die Liste $\{1, 15, 35, 45\}$ würde zu $\{1, 14, 20, 10\}$) kann man kleinere Durchschnittswerte enthalten. Da diese jedoch auch sehr große Zahlen enthalten können wäre man gezwungen auch weiterhin im Beispiel 17 Bit zu verwenden. Jedoch da man nur selten große Zahlen hat lohnt es sich variable Bit raten zu verwenden.

Dazu gibt es einerseits parameterlose Codes wie Unary oder die Elias Gamma und Delta Codes.

- **Unary** Eine Zahl x wird codiert durch x-1 Einsen gefolgt von einer Null. Obwohl dies Verschwenderisch kling kann man beweisen das diese Kodierung Optimal ist wenn 50% aller Zahlen Einsen sein 25% Zweien usw. Diese Kodierung eignet sich somit sehr gut für Terme welche in fast jedem Dokument vorkommen.

- **Elias Gamma** Repräsentiert x als $2^e + d$ durch Kodierung von $e + 1$ in unary und d in e Bits binär Kodierung.
- **Elias Delta** Gleich wie Elias Gamma jedoch wird e+1 wiederum in der Gamma Kodierung codiert.

value	unary	gamma	delta
1	0	0:	0::
2	10	10:0	10:0:0
3	110	10:1	10:0:1
4	1110	110:00	10:1:00
10	1111111110	1110:010	110:00:010
100		1111110:100100	110:11:100100
1,000		1111111110:111101000	1110:010:111101000

Abbildung 3: Beispiel Parameterlose Codes

Während die Elias Gamma und Delta Kodierung sich für die Positionen in Dokumenten bewährt hat. So haben sich speziell für die Dokumenten IDs der Invertieten Listen jedoch in Experimenten parametrisierte Codes wie die Golomb Codierung als Effizient erwiesen. Diese Kodierungen verwendet einen Parameter um zu bestimmen in welchem Bereich die Codierung sehr sparsam ist. Und könne so den Gegebenheiten gut angepasst werden. Die Golomb Kodierung kodiert dabei eine Zahl x als

$q * b + r + 1(r < b)$

Und Kodiert $q + 1$ als Unary. Berechnet 2 Hilfswerte

$e = \lceil log_2 b \rceil \qquad g = 2^e – b$

Wenn $r < g$ wird r binär kodiert mit $e - 1$ Bits ansonsten wird $g + r$ Kodiert in e Bits.

value	$b = 3$	$b = 5$	$b = 16$
1	0:0	0:00	0:0000
2	0:01	0:01	0:0001
3	0:11	0:10	0:0010
4	10:0	0:110	0:0010
10	1110:0	10:110	0:1001

Abbildung 4: Golomb Beispiel

Diese Kompression der Listen hat aber nicht nur Platzvorteile. Indirekt wird so der Zugriff auch schneller da sich durch den kleineren Plattenplatz das Caching und die Lesezeit verbessert. Dies überwiegt die CPU Zeit für die Dekodierung der Listen. Ein Nachteil der noch behandelt werden soll ist das Updates durch die Kompression erschwert werden.

4.2 2-Stufiger lookup

Eine weitere Technik um einerseits den Platzbedarf weiter zu senken und zudem besser Boolsche AND Anfragen beantworten zu können ist der 2-Stufige Lookup. Hierbei wird die ebene Invertierte Liste durch eine 2- Stufige Liste ersetzt. Man unterteilt die liste in k Eimer mit ausreichender Größe ([2] 500 Einträge pro Eimer) und gibt für jeden Eimer den Ersten wert an sowie einen eigenen Wert für die Golomb Kodierung. Durch diese individuelle Anpassung wird weiter platz gespart.

Zudem erlaubt es begrenzt binär suchen auf der Liste auszuführen da man auf der Oberen ebene suchen kann und nur einen Teil der Liste dekodieren muss. Ansonsten hat man nur die Möglichkeiten den Zipper algorithmus zu verwenden d.h. um ein AND zu bilden durch beide Listen durch zugehen und immer bei der Liste die momentan an der kleineren ID ist die nächste ID zu betrachten. Diese Methode ist sehr effektiv wenn beide Listen gleich groß sind. Ist jedoch eine der Listen relativ klein ist eine Binärsuche aller Elemente aus der kleineren Liste in der größeren Liste effizienter. vgl Zipper: $O(n+m)$, Binärsuche: $O(n*log(m))$ für $n << m$ schneller.

4.3 Update Strategien

Je nach Szenario verändert sich der indexierte Text z.B. findet ein webcrawler neue Seiten für eine Suchmaschine. Dementsprechend muss der Index upgedatet werden. Dies ist jedoch nicht Trivial, da ein einzelnes neues Dokument einzufügen bedeuten würde hunderte Invertierte Listen von der Festplatte zu lesen zu dekomprimieren und die neue Dokumenten ID einzufügen. So gibt es 3 bekannte Strategien die Anwendung finden [3].

- **Rebuild** bedeutet einfach in regelmäßigen Abständen den Index komplett neu Aufzubauen. Jedoch fordert dies das der doppelte Speicher zur Verfügung steht um den alten Index parallel zu halten.
- **Intermittent merge** bildet mit neu ankommenden Dokumenten eine neuen Index in der parallel gesucht wird. Diese wird nach einer gewissen Zeit oder Größe mit dem alten Index verschmolzen.
- **Incremental Update** ist eine Lazy Update Methode. Hierbei werden neue Dokumenten IDs für die Invertierten Listen im Speicher gehalten. Wann immer eine invertierte Liste z.B. durch eine Suche geöffnet wird werden die neuen IDs in diese Liste eingefügt.

4.4 Verteilte Anfragen

Wenn ein Index größer wird und die Zahl der Nutzer Steigt kann ein einzelner Rechner nicht mehr alle Daten halten und auch nicht mehr alle Anfragen beantworten. Also benötigt man Strategien um diese Last zu verteilen.

- **Term Distribution** Dabei werden Rechnern Termen zugeteilt. Der Rechner hält dann alle Dokumente die einen Term enthalten.
- **Dokument Distribution** Dabei werden Rechnern Dokumente zugeteilt. Der Rechner baut einen Index über die ihm zugeteilte menge der Dokumente auf.

- **Replikation** Ein Rechner der einen Teilindex hält wird einfach dupliziert. So kann man anfragen aufteilen und de Last pro Rechner sinkt. Dies Fördert auch die Ausfallsicherheit.

Alle diese Strategien werden vermischt eingesetzt um einen zuverlässigen schnellen Index aufzubauen.

5 Abschließende Bemerkungen

Der Invertierte Index eine effiziente Suchstruktur die überall wo natürlich Sprachlicher Text durchsucht wird seine Anwendung hat. Mit Fallenden kosten für den Speicher und besseren Techniken die die Größe weiter drücken wird es mittlerweile sogar möglich sie evtl komplett im Speicher zu halten. Größen von 0.57n [2] sind bereits machbar. Damit ist der Index zwar nur minimal kleiner wie ein komprimiertes Suffx Array kommt jedoch ohne den dort zur Erstellung benötigten Speicheroverhead aus.

Literatur

[1] S. Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 29(13):1149–1171, 1999.

[2] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 222–229, New York, NY, USA, 2002. ACM.

[3] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.