Top-K Retrieval

Seminar Information Retrieval SoSe 2012

Martin Leinberger

Universität Koblenz-Landau,
Universitätsstraße 1, 56070 Koblenz, Deutschland
mleinberger@uni-koblenz.de

Abstract. Moderne Suchmaschinen zeigen in der Regel nur einen Bruchteil der gefundenen Ergebnisse an. Auch wenn Tausende von Treffern gefunden wurden, liefert das System oft vorerst nur zehn Ergebnisse. Insbesondere im World Wide Web in dem Millionen von potentiellen Ergebnissen vorhanden sind und Effizienz ein wichtiges Kriterium ist, sind Algorithmen sehr wichtig die die Top-K Treffer, also die k relevantesten Dokumente, möglichst schnell bestimmen können. Diese Arbeit stellt einige bekannte Algorithmen aus diesem Gebiet vor und erläutert ihre Vorgehensweise.

1 Einleitung

Moderne, im Web genutzte Suchmaschinen zeigen ihre Ergebnisse in kleinen Teilen an. Sie unterteilen die von ihnen gefundenen Ergebnisse in Seiten um dem Nutzer eine bessere Übersicht zu ermöglichen. So zeigt beispielsweise Google bei einer Websuche zehn Ergebnisse auf der ersten Seite an. Möchte der Nutzer weitere Ergebnisse sehen, so muss er dies explizit anfordern, indem er auf die zweite Seite wechselt.
Allerdings kann das Informationsbedürfnis des Nutzers oft schon durch eine einzige relevante Quelle befriedigt werden. Sollten sich auf der ersten Seite keine relevanten Quellen befinden, so wird er meistens einfach die Anfrage neu formulieren, da es unwahrscheinlich ist, dass er auf der zweiten Seite ein relevantes Ergebnis findet. Der Nutzer wechselt also sehr selten auf die zweite Seite der Suchergebnisse. In der Praxis sind also eher die k relevantesten Dokumente gefordert, statt alle relevanten Dokumente. Da die Zufriedenheit des Nutzers in hohem Maße mit der Geschwindigkeit der Suchmaschine einhergeht, stellt sich also die Frage wie man möglichst effizient diese Top-K Dokumente bestimmen kann.

2 Aufbau der Arbeit

Um die in der Einleitung präsentierte Fragestellung zu beantworten wird in dieser Arbeit zuerst kurz auf das im IR übliche Datenmodell eingegangen (Abschnitt 3). Anschließend werden erste naive Algorithmen präsentiert (Abschnitt 4), bevor im

adfa, p. 1, 2011.

Hauptteil der Ausarbeitung auf verschiedene Top-K Algorithmen eingegangen wird (Abschnitt 5). Bei allen Algorithmen wird zuerst die Grundidee erläutert, dann formaler auf ihn eingegangen, kurz die Korrektheit erläutert und an einem einheitlichen Beispiel verdeutlicht. Abschließend wird ein kurzes Fazit gezogen (Kapitel 6).

3 Datenmodell

IR Systeme dienen dazu Informationsobjekte zu finden. Sehr oft sind diese Informationsobjekte Dokumente. Von diesen Dokumenten sind hauptsächlich die darin enthaltenen Terme interessant. Um die Dokumente und ihre Terme zu verwalten wird im Information Retrieval hauptsächlich eine Datenstruktur namens Invertierter Index eingesetzt. Dieser ist prinzipiell ein Suchbaum. Für jeden Term, der in einem Dokument des Systems vorkommt wird eine sogenannte Postingliste angelegt. Diese enthält Referenzen auf Dokumente. Es werden genau die Dokumente in ihr abgelegt, in denen der Term, zu dem sie gehört, enthalten ist. Die Terme, sowie ihre zugehörigen Postinglisten werden dann in den Invertierten Index einsortiert. Prinzipiell bildet der Invertierte Index also Terme auf Listen von Dokumentreferenzen ab.
In der Praxis sind die Einträge in den Postinglisten oft attributiert. Meistens speichert man Gewichte zusammen mit den Dokumentreferenzen ab, die eine Aussage darüber machen, wie wichtig der Term in diesem Dokument ist. Ein übliches Beispiel für ein solches Gewicht ist die Termfrequenz. Zusätzlich können die Postinglisten nach verschiedenen Werten, wie Dokumenten-Id oder Gewichten, sortiert sein. Ein Beispiel für einen Ausschnitt aus einem Invertierten Index ist in Fig. 1 dargestellt.

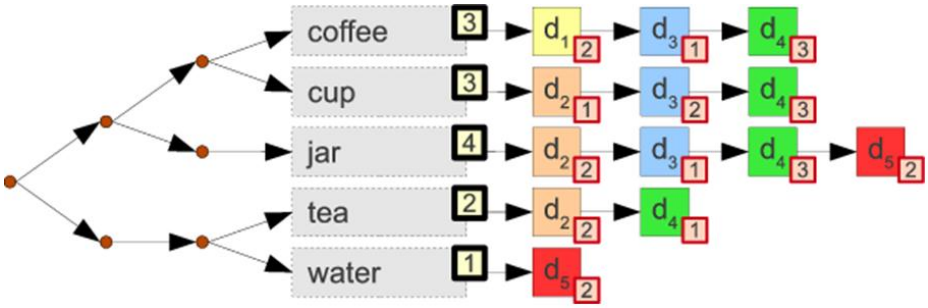

Fig. 1. Invertierter Index als Suchbaum mit eingetragenen Termen sowie Postinglisten sortiert nach Dokumenten-Id. Die Postinglisten sind mit Termfrequenzen attributiert, die Terme mit Dokumentfrequenzen. (Quelle: [5]).

Die vom System genutzte Retrievalfunktion dient dazu die „Relevanz“ von Dokumenten bezüglich einer Anfrage zu berechnen. Dazu errechnet sie einen Retrievalwert, der an dieser Stelle mit der Relevanz gleichgesetzt wird. Das Dokument mit dem höchsten Retrievalwert wird auch als das relevanteste angesehen. Um den Retrievalwert zu errechnen, aggregiert die Retrievalfunktion die in den Postinglisten abgespeicherten Werte zu einem Gesamtwert. Dazu können verschiedene Funktionen genutzt werden. Übliche Beispiele für diese sind BM-25 oder auch das Kosinusmaß.

Welche Funktion tatsächlich genutzt wird, ist für diese Ausarbeitung unwichtig. Wichtig ist jedoch das die Retrievalfunktion die Monotonieforderung erfüllt, also für eine Funktion $f(x_1, \dots, x_m)$ gilt dass $f(x_1, \dots, x_m) \leq f(x'_1, \dots, x'_m)\ wenn\ x_i \leq x'_i$ für jedes i . Umgangssprachlich bedeutet dies, dass der Retrievalwert eines Dokuments kleiner als der eines anderen Dokuments sein sollte, falls alle Termgewichte des Dokuments kleiner sind. Für übliche Retrievalfunktionen, wie oben beispielsweise benannt, gilt dies.
Das Ergebnis einer Anfrage ist nun eine Teilmenge der Dokumente. Genauer gesagt ist es eine Liste aus Tupeln, wobei ein Tupel aus einer Referenz auf ein Dokument sowie dem Retrievalwert des Dokuments besteht. Es wird also für jedes Dokument in der Ergebnisliste auch der Retrievalwert gefordert[1]. Üblicherweise benutzt man einen Grenzwert und schneidet die Ergebnisliste ab diesem Grenzwert ab. Im Top-K Retrieval wäre dieser Grenzwert genau der Retrievalwert des k-ten Ergebnisses. Jedoch ist dieser Wert im Normalfall unbekannt.

4 Naive Algorithmen

4.1 Term-at-a-Time

Nach [3] ist eine der im Information Retrieval üblichen Vorgehensweisen das sogenannte Term-at-a-Time Verfahren. Dabei wird jeder Term bzw. Teil der Anfrage nacheinander bearbeitet. Die Liste des Terms wird also erst vollständig abgearbeitet, bevor der nächste Term betrachtet wird. Die entstehenden Zwischenergebnisse für einzelne Dokumente werden gespeichert und stückchenweise aktualisiert. Die so entstehende Ergebnisliste muss später sortiert werden, damit die Top-K Ergebnisse angezeigt werden können.
Ein großer Nachteil an einem Term-at-a-Time Verfahren ist, dass die Liste für einen Term komplett durchlaufen werden muss (siehe [3]). Diese Listen können sehr lang werden. Außerdem muss für jedes dabei auftauchende Dokument eine Variable zum Speichern der Zwischenergebnisse angelegt werden. Auch die Ergebnisliste, welche später sortiert werden muss, kann sehr lang werden.

4.2 Document-at-a-Time

Ein Document-at-a-Time Verfahren ist eine andere, im IR übliche Vorgehensweise (siehe [3]). Im Gegensatz zu Term-at-a-Time wird der Relevanz-Wert eines Dokuments erst vollständig berechnet, bevor das nächste betrachtet wird. Dabei wird ausgenutzt das die Listen für die Terme nach Dokumenten-Ids sortiert sein können. Durchläuft man sie parallel kann ein Document-at-a-Time Verfahren effizient implementiert werden.
Die Listen müssen allerdings wieder vollständig durchlaufen werden. Eine Verbesserung ist jedoch das lediglich k Variablen benötigt werden. Sobald der Wert eines neu-

[1] Was logisch erscheint, da ja die relevantesten Dokumente auch an erster Stelle stehen sollten.

en Dokuments berechnet wurde, kann verglichen werden ob der Wert höher als das bisherige Minimum der Top-k Dokumente ist. Falls ja, kann es dieses ersetzen. Am Ende des Retrieval Vorgangs muss nur die Liste der Top-k Dokumente sortiert werden, was eine erheblich kürzere Liste darstellt als die aller betrachteten Dokumente.

4.3 Probleme

Das Hauptproblem einer naiven Herangehensweise ist, dass die Listen vollständig durchlaufen werden müssen. Der Dokumentenkorpus kann sehr groß werden und ein Term kann in sehr vielen Dokumenten vorkommen. Insbesondere in IR Systemen, die im World Wide Web eingesetzt werden, kann die Liste für einen Term so lang werden, dass es nicht mehr praktikabel ist, sie vollständig zu durchlaufen. Auch das Anlegen einer Variablen für jedes beobachtete Dokument wird bei sehr vielen Dokumenten problematisch. Benötigt wird ein Mechanismus, der es dem Retrievalsystem erlaubt, den Prozess möglichst früh abzubrechen.

5 Top-K Algorithmen

5.1 Einleitung

Sogenannte Multimedia Datenbanken stehen vor den gleichen Herausforderungen wie das IR und nutzen dabei ähnliche Datenstrukturen. Multimedia Datenbanken vereinen unterschiedliche Datenquellen wie Videos oder Bilder. Dazu vergeben sie gewisse Eigenschaften für die Objekte aus diesen Datenquellen. Ein Beispiel wäre eine Eigenschaft, die aussagt wie „Rot“ ein Bild ist. Die Eigenschaft „Rot“ hat eine Liste aus Tupeln mit ihr assoziiert, die für alle Objekte die die Eigenschaft „Rot“ besitzen angibt, wie stark diese Eigenschaft erfüllt ist (siehe [1]). Die Datenstruktur gleicht also dem Invertierten Index. Daher lassen sich die in ihrem Kontext entwickelten Algorithmen problemlos auch im IR anwenden.
Die Algorithmen basieren dabei auf einigen grundsätzlichen Beobachtungen. Die Listen dürfen nicht mehr komplett gelesen werden. Um dies zu verhindern, ist es nötig, Objekte mit hohen Gewichten möglichst früh zu betrachten. Die mit den Eigenschaften bzw. den Termen assoziierten Listen sollten also nach dem Gewicht und nicht nach einer Objekt-Id sortiert sein. Denn dann können, unter der Annahme, dass die Retrievalfunktion bzw. die Aggregationsfunktion monoton ist, verschiedene Punkte gefunden werden, an denen es sicher ist, dass keine besseren Dokumente mehr gefunden werden können.
Beim Zugriff auf einzelne Dokumente bzw. der Objekte der Multimedia DB kann zwischen zwei Möglichkeiten unterschiedenen werden (siehe [1]. Zum einen kann sequentiell auf die Daten zugegriffen werden (Sequential Access). Dabei wird auf einen Eintrag einer Liste zugegriffen bzw. es werden Einträge hintereinander aus einer Liste gelesen. Diese Art des Zugriffs gilt im Allgemeinen als sehr schnell. Kosten, die durch Lesezugriffe auf Datenträger wie Festplatten anfallen, amortisieren sich bei vielen Zugriffen. Die zweite Art des Zugriffs ist hingegen der Random Access. Dabei wird das Gewicht für einen speziellen Eintrag nachgeschaut. Diese Art des

Zugriffs ist sehr teuer. Neben den Kosten für das Nachschauen der Position des Eintrags müssen auch die vollen Kosten für das Lesen der Daten vom Datenträger bezahlt werde. Laut [3] ist ein Random Accesses 10 – 1000 mal langsamer, als ein Sequential Access.

5.2 Fagin's Algorithm

Der Algorithmus von Fagin ist einer der bekanntesten Algorithmen im Top-K Retrieval. Er wurde beispielsweise in IBMs experimentellen Datenbanksystem Garlic implementiert (siehe [1]). Der Algorithmus ist in 3 Phasen unterteilt. In der ersten Phase sammelt er alle Dokumente die theoretisch in Frage kommen könnten durch sequentielle Zugriffe. Er bricht diese Phase ab, sobald er k Dokumente gefunden hat, die in allen, für die Anfrage wichtigen, Listen vorkommen. Er bricht sie also ab, sobald k Dokumente vollständig bekannt sind. Die restlichen, bisher nur teilweise bekannten Dokumente werden in Variablen gespeichert. Dazu benötigt er jedoch theoretisch beliebig viele Variablen. Er fängt dann an, die von ihm gefundenen, noch nicht vollständigen Ergebnisse, mithilfe von Random Accesses zu vervollständigen. Das Ergebnis ist nun eine kleine Teilmenge des Korpus, für den alle Gewichte bekannt sind. Für diesen kann nun der Retrieval-Wert aller Dokumente berechnet werden und die Top-K Dokumente herausgesucht werden.
Formaler kann man den Ablauf des Algorithmus von Fagin folgendermaßen beschreiben (siehe [1]):

1. Greife parallel[2] auf jede der *m* Listen zu, die zum Relevanzwert bei der gegebenen Anfrage beitragen. Füge die gefundenen Dokumente zur Menge *H* hinzu. Falls ein Dokument noch nicht bekannt ist, lege eine Variable an. Ansonsten erweitere die bestehende Variable um das neu gefundene Gewicht. Wiederhole dies so lange bis sich in *H* k Dokumente befinden, die in allen *m* Listen gefunden wurden (also bis k Dokumente vollständig bekannt sind).
2. Greife mithilfe von Random Accesses für jedes Dokument *R* aus der Menge *H* auf die noch fehlenden Termgewichte für *R* zu und vervollständige so die Gewichte für das Dokument.
3. Berechne den finalen Relevanzwert für jedes Dokument aus *H*. Sei *Y* die Menge der k Objekte aus *H* mit den höchsten Relevanzwerten. Das Ergebnis des Algorithmus ist dann eine sortierte Liste aus Tupeln $\{(R, f(R))|R \in Y\}$.

Dass dieser Algorithmus das korrekte Ergebnis liefert ist relativ einfach aus der Monotonieforderung ersichtlich. Angenommen ein Dokument *Z*, das sich nicht in *H* befindet, hat einen höheren Retrievalwert als ein Dokument *R* aus *H*. Da allerdings *Z* sich nicht in *H* befindet, müssen alle Termgewichte x_i kleiner als die Termgewichte x_i' des Dokuments *R* sein. Aus der Monotonie der Retrievalfunktion folgt damit $f(Z) \leq f(R)$.

---

[2] Parallel bedeutet in diesem Fall eigentlich „in lockstep". Also ein Zugriff auf das erste Element der ersten Liste, dann auf das erste Element der zweiten Liste, usw.

Das Verfahren kann anhand eines Beispiels verdeutlicht werden. Gesucht werden in diesem Beispiel die Top-2 Dokumente. Es existieren zwei Listen, die für die Anfrage relevant sind. Als Retrievalfunktion wird die Summen-Funktion genutzt, die einfach die Termgewichte aufsummiert.
In den ersten beiden Schritten des Algorithmus werden mit parallelen Zugriffen die beiden Listen durchlaufen. Die neu gefundenen Dokumente a, d und b werden in Variablen gespeichert, das im zweiten Schritt bereits bekannte Dokument a wird aktualisiert (siehe Fig. 2).

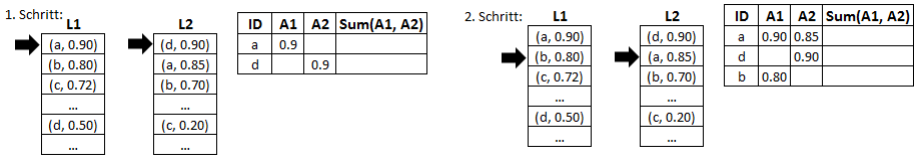

Fig. 2. Die ersten beiden Schritte des Algorithmus von Fagin für das gegebene Beispiel.

Im nächsten Schritt (Schritt 3) werden c und b gefunden. Damit sind nach diesem Schritt a und b vollständig bekannt. Die Top-K Dokumente müssen also in den bisher gefundenen Dokumenten liegen. Mithilfe von Random Accesses werden in Schritt 4 die fehlenden Termgewichte für d und c gefunden, die finalen Retrievalwerte berechnet und die Top-2 (Dokumente a und b) zurückgegeben (siehe Fig. 3).

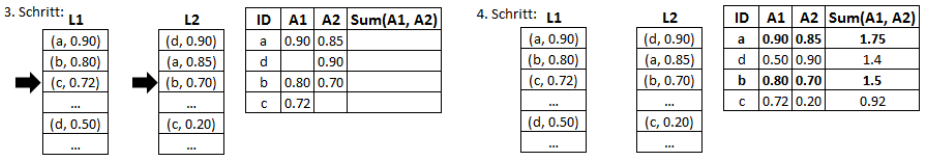

Fig. 3. Die letzten beiden Schritte des Algorithmus von Fagin für das gegebene Beispiel.

5.3 Threshold Algorithm

Der Threshold Algorithm basiert auf der Idee, dass ein Grenzwert für die noch unbetrachteten Dokumente existieren muss. Haben alle Dokumente in den Top-K diesen Grenzwert überschritten, ist es, wie [1] erläutert, unmöglich, dass ein anderes Dokument besser ist, als die, die sich derzeit in den Top-K befinden. Dieser Grenzwert lässt sich aus den beobachteten Minima aus den einzelnen Listen ableiten. Der tiefste, in einer Liste beobachtete Wert stellt eine obere Grenze für die Werte in dieser Liste dar, da alle darauffolgenden Werte kleiner oder gleich diesem Wert sein müssen. Dies folgt aus der Sortierung der Liste. Aggregiert man alle beobachteten oberen Grenzen mithilfe der Retrievalfunktion, bekommt man den Wert, den die nächsten Dokumente maximal haben können. Haben die Top-K Dokumente also einen größeren Wert so kann der Algorithmus terminieren. Dabei müssen zu jedem Zeitpunkt lediglich die k besten Dokumente gespeichert werden, es wird also nur eine feste Anzahl an Variablen benötigt.

Formaler kann der Algorithmus folgendermaßen definiert werden (siehe [1]):

1. Greife parallel auf jede der *m* Listen zu, die zum Relevanzwert bei der gegebenen Anfrage beitragen. Wenn ein Dokument *R* zum ersten Mal beobachtet wird, greife mithilfe von Random Accesses auf die fehlenden Termgewichte zu um den Relevanzwert von *R* vollständig berechnen zu können. Berechne den Relevanzwert von R und speichere R sowie $f(R)$, falls es zu den k besten Dokumenten gehört.
2. Sei für jede Liste L_i der Wert $\underline{x_i}$ das Gewicht des zuletzt gesehenen Dokuments der Liste. Dann ist der Grenzwert $\tau = f(\underline{x_1}, .., \underline{x_m})$. Sobald k Dokumente gefunden wurden, deren Relevanzwert größer oder gleich τ ist terminiere. Ansonsten expandiere die Listen weiter (siehe 1).
3. Sei Y die Menge der k Dokumente mit den höchsten Relevanzwerten. Die Ausgabe des Algorithmus ist dann die sortierte Liste $\{(R, f(R)\}|R \in Y)\}$.

Um die Korrektheit zu zeigen, muss lediglich gezeigt werden dass der Relevanzwert jedes Dokuments *y* das sich in Y befindet, größer oder gleich den Werten jedes Dokuments z ist, das sich nicht in Y befindet. Das *y* größer ist als *z*, falls *z* betrachtet wurde, ist offensichtlich. Falls *z* noch nicht betrachtet wurde, muss gezeigt werden, dass der Relevanzwert von *z* kleiner oder gleich dem Grenzwert τ ist. Da *z* nicht betrachtet wurde, muss aber für jedes Termgewicht von x_i von *z* gelten, dass $x_i \leq \underline{x_i}$. Dadurch muss aber durch die Monotonieforderung auch gelten, dass $f(z) \leq \tau \leq f(y)$.
Das Beispiel verdeutlicht das Vorgehen. Im ersten Schritt werden dabei wieder die Dokumente a und d gefunden. Die fehlenden Termgewichte werden sofort mithilfe von Random Accesses gefunden. Der Grenzwert ist das bisherige Minimum aus jeder Liste, also aus der Summe von 0.9 und 0.9 (siehe Fig. 4).

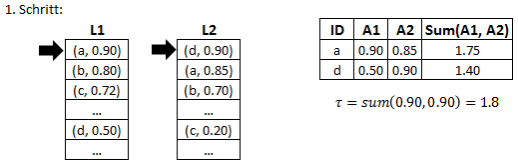

Fig. 4. Erster Schritt des Threshold-Verfahrens für das gegebene Beispiel.

Im nächsten Schritt wird die Liste weiter expandiert. Das Dokument d wird aus den Top-2 herausgenommen und durch b ersetzt, da der Retrievalwert für b höher ist.

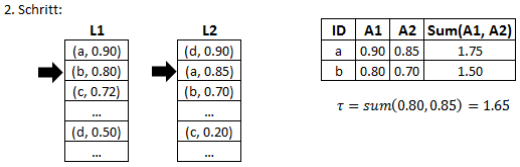

Fig. 5. Zweiter Schritt des Threshold Verfahrens für das gegebene Beispiel.

Nach der nächsten Expansion verändert sich an den Top-2 Dokumenten nichts, da der Retrievalwert von c zu klein ist. Jedoch ist der Grenzwert nun kleiner als das minimum aus den Top-2 Dokumenten. Der Algorithmus kann also nun terminieren.

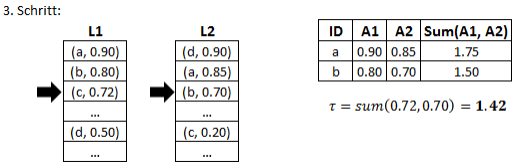

Fig. 6. Dritter Schritt des Treshold Verfahrens für das gegebene Beispiel

5.4 Random Accesses

Random Accesses stellen eine gewisse Herausforderung bei dem Design eines Algorithmus dar. Sie werden oft genutzt um fehlende Eigenschaften für ein Dokument zu erhalten. Dies ist notwendig um den finalen Retrievalwert zu errechnen. Der optimale Zeitpunkt für einen Random Access ist im Allgemeinen nicht bekannt. Die fehlende Eigenschaft könnte bereits bei einer der nächsten Expansionen einer Liste zu finden sein. In diesem Fall wäre der Random Access verschwendet. Andererseits kann es jedoch auch sein, dass die Eigenschaft erst nach 1000 weiteren Expansionen gefunden wird.
Eine übliche Maßnahme ist es einen Random Access erst nach *C* sequentiellen Zugriffen zu erlauben und sie so zu beschränken. [3] argumentiert jedoch, dass dadurch trotzdem viele Random Accesses verschwendet werden. Eine weitere, von [3] vorgestellte Möglichkeit ist die LAST Heuristik, bei denen Random Accesses erst erlaubt werden, nachdem alle möglichen Top-K Dokumente gefunden wurden[3] und die erwarteten Kosten für den Random Access unter den bisher für sequentielle Zugriffe benutzten Kosten liegen. Dies ist der Versuch Random Accesses so spät wie möglich zu verwenden um zu verhindern, dass man sie verschwendet.
Jedoch gibt es auch die Möglichkeit, dass das System Random Accesses gar nicht erlaubt. Aus diesem Grund sollen im Folgenden zwei weitere Algorithmen vorgestellt werden die sich genau dieser Problematik annehmen und ganz auf Random Accesses verzichten.
Dabei kommen zwei unterschiedliche Philosophien zum Einsatz. Zum einen kann darauf verzichtet werden, den finalen Retrievalwert zu berechnen. Dadurch können Eigenschaften unbekannt bleiben und man fordert eher die Menge der Top-K Dokumente als eine sortierte Liste. Die zweite Möglichkeit ist es, solange mithilfe von Sequentiellen Zugriffen die Liste zu expandieren, bis die fehlenden Eigenschaften gefunden sind.

[3] Natürlich sind sie zu dem Zeitpunkt noch nicht vollständig bekannt, es ist lediglich sicher dass diese Dokumente die Top-K darstellen.

5.5 No Random Accesses (NRA) Algorithm

Der NRA Algorithmus ist eine Abwandlung des Threshold Algorithmus. Er benutzt im Gegensatz zum Threshold Algorithm keinerlei Random Accesses. Allerdings fordert der NRA Algorithmus auch ein schwächeres Ergebnis. Während die in 5.2 und 5.3 vorgestellten Algorithmen eine Liste von Tupeln mit einer Referenz auf das Dokument sowie dem Retrievalwert des Dokuments zurück liefern, liefert der NRA lediglich eine Menge von Dokumenten ohne Relevanzwerte. Dies kommt daher, dass der NRA Algorithmus den Relevanzwert lediglich soweit berechnet, bis er sich sicher sein kann, dass er terminieren kann. Dies muss jedoch nicht der vollständige Retrievalwert sein. [1] weist jedoch darauf hin, dass es dem NRA trotzdem möglich ist, ein Ranking zu erstellen, indem man zuerst nach den Top-1 Dokumenten sucht, danach nach den Top-2 usw.
Der NRA Algorithmus nutzt einen best score und einen worst score. Der worst score ist der Relevanzwert der bekannten Termgewichte für ein Dokument, wobei bisher noch unbekannte Gewichte den Wert 0 bekommen. Der best score hingegen setzt für die unbekannten Gewichte das beobachtete Minimum der Liste als obere Grenze ein. Dieser best score stellt den maximalen Retrievalwert dar, den das Dokument erreichen kann. Dementsprechend ist das Abbruchkriterium des Algorithmus, wenn der worst score der Top-K Dokumente besser ist als der best score der anderen, beobachteten Dokumente.
Der NRA Algorithmus lässt sich folgendermaßen formaler definieren (siehe 1):

1. Greife parallel auf jede der *m* Listen zu, die zum Relevanzwert bei der gegebenen Anfrage beitragen. Lege für neu gefundene Dokumente eine Variable an oder aktualisiere den Wert für das Termgewicht falls das Dokument bereits bekannt ist.
2. Sei für jede Liste L_i der Wert $\underline{x_i}$ das Gewicht des zuletzt betrachteten Dokuments. Berechne den worst score eines jeden Dokuments R, wobei für jedes unbekannte Termgewicht von R 0 eingesetzt wird. Berechne nun den best score von R, wobei für jedes unbekannte Termgewicht der Wert von $\underline{x_i}$ eingesetzt wird.
3. Seien die Top-K Dokumente die k Dokumente mit dem besten worst score. Terminiere falls der worst score aller Top-K Dokumente besser ist als der best score aller anderen, bisher betrachteten Dokumente und gebe die Top-K Dokumente zurück. Ansonsten expandiere die Listen weiter (siehe 1).

Die Korrektheit erfolgt erneut aus der Monotonie. Das Ergebnis des Algorithmus ist die Menge $T_K = \{R_1, R_2, \dots, R_k\}$. Angenommen ein Dokument R ist nicht in dieser Menge. Da der Algorithmus terminierte wissen wir, dass der best score $b(R)$ kleiner gleich dem worst score der Dokumente in T_K sein muss. Also gilt $b(R) \leq w(R_i)$. Es ist jedoch auch bekannt, dass der best score obere Grenzen der Listen enthält, also immer $b(R) \geq f(R)$ gilt. Gleichermaßen gilt für den worst score, dass $w(R) \leq f(R)$ sein muss, da dieser unbekannte Termgewichte mit dem Wert 0 besetzt. Daraus folgt, dass $f(R) \leq w(R_i) \leq f(R_i)$ $für\ jedes\ R_i \in T_K$.
Das Verfahren lässt sich erneut durch ein Beispiel verdeutlichen. Im ersten Schritt werden die Dokumente a und d gefunden, sowie ihr best und worst score berechnet (siehe Fig. 7).

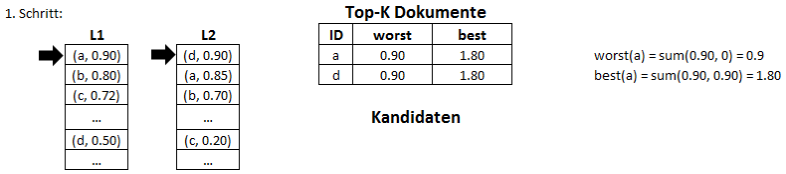

Fig. 7. Erster Schritt des NRA Verfahrens für das gegebene Beispiel.

Nach dem nächsten Schritt ist a vollständig bekannt, b wird neu gefunden. Der best score für d wird aktualisiert (siehe Fig. 8).

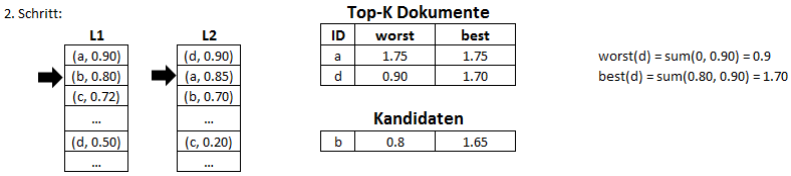

Fig. 8. Zweiter Schritt des NRA Verfahrens für das gegebene Beispiel.

Nach dem dritten Schritt ist auch d vollständig bekannt, jedoch muss der Algorithmus weiter machen. Der best score von d zeigt, dass d theoretisch einen höheren Retrievalwert als b haben könnte (siehe Fig. 9).

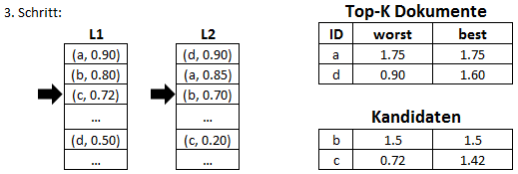

Fig. 9. Dritter Schritt des NRA Verfahrens für das gegebene Beispiel.

Nachdem eine Reihe weiterer Dokumente betrachtet wurde, kommt der Algorithmus zu dem Punkt, an dem sich der best score von d soweit verschlechtert hat, dass er kleiner als der worst score von b wird. Nun kann der Algorithmus terminieren (siehe Fig. 10).

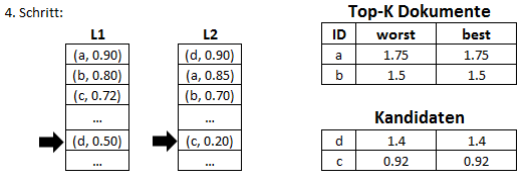

Fig. 10. Vierter Schritt und Ende des NRA Verfahrens für das gegebene Beispiel.

5.6 Stream Combine

Der Stream Combine Algorithmus ist ein weiterer Algorithmus der komplett auf Random Accesses verzichtet. Die von [2] vorgestellte Idee ist, so lange mit sequentiellen Zugriffen durch die Listen zu wandern, bis k Dokumente ausgegeben wurden. Dabei wird in jedem Schritt der Retrievalwert berechnet. Sind Werte des Dokuments noch unbekannt, werden sie auf das bisherige Minimum aus der entsprechenden Liste gesetzt und dienen so wieder als obere Grenze. Ein Dokument kann ausgegeben werden, falls es kein anderes Dokument mehr gibt, dessen finaler Retrievalwert noch unbekannt ist und dessen geschätzter Wert höher ist. Nachdem ein Dokument ausgegeben wurde kann die Variable, die dieses Dokument repräsentiert gelöscht werden.
Formaler gesehen lässt sich der Stream Combine Algorithmus folgendermaßen beschreiben (siehe [2]):

1. Greife parallel auf jede der *m* Listen zu, die zum Relevanzwert bei der gegebenen Anfrage beitragen. Für jedes neu gefundene Dokument *R*, wird eine Variable angelegt und der neu gefundene Wert gespeichert. Unbekannte Werte werden mit *NULL* initialisiert. Für bereits bekannte Dokumente wird der neu gefundene Wert übernommen.
2. Sei für jede Liste L_i der Wert $\underline{x_i}$ das Gewicht des zuletzt in dieser Liste gesehenen Dokuments. Für jedes Dokument *R* das nur teilweise bekannt ist werden die unbekannten Werte auf das entsprechende $\underline{x_i}$ gesetzt. Dies dient als obere Grenze für den Retrievalwert des Dokuments *R*.
3. Für jedes Dokument *R*, das bisher beobachtet wurde, wird der Retrievalwert $f(R)$ berechnet.
4. Falls ein Dokument R_{Top} in allen Listen gefunden wurde, also vollständig bekannt ist, muss getestet werden, ob es ein Dokument mit einem höheren Retrievalwert gibt. Falls nicht kann das Dokument R_{Top} als nächstes der k besten Dokumente zusammen mit seinem Retrievalwert ausgegeben werden und seine Variable gelöscht werden. Wurden k Dokumente ausgegeben, terminiert der Algorithmus. Gibt es jedoch noch ein Dokument mit einem höheren Retrievalwert, kann R_{Top} noch nicht ausgegeben werden und es müssen erneut die Listen expandiert werden (siehe 1).

Dieses Vorgehen liefert erneut aufgrund der Monotonie das korrekte Ergebnis. Angenommen ein Dokument *z* hätte einen größeren Retrievalwert als das zuletzt ausgegebene Dokument R_{Top}. Da R_{Top} ausgegeben wurde, war sein Retrievalwert größer als der aller bisher betrachteten Dokumente. Wurde *z* noch nicht betrachtet, so muss für jedes Gewicht x_i von *z* gelten, dass es kleiner ist als jedes Gewicht x_i' von R_{Top}. Daraus folgt dass $f(z) \leq f(R_{Top})$ sein muss. Wurde *z* jedoch betrachtet, muss auch in diesem Fall der Retrievalwert kleiner als der von R_{Top} sein, da $\underline{x_i}$ die obere Grenze darstellt und somit gilt, dass $\underline{x_i} \geq x_i$ ist. Also muss auch in diesem Fall folgen, dass $f(z) \leq f(R_{Top})$ ist.
Durch ein Beispiel kann der Algorithmus verdeutlicht werden. Im ersten Schritt werden die Dokumente a und d gefunden. Für die fehlenden Eigenschaften werden die

Werte der zuletzt betrachteten Dokumente aus der jeweiligen Liste eingesetzt (siehe Fig. 11).

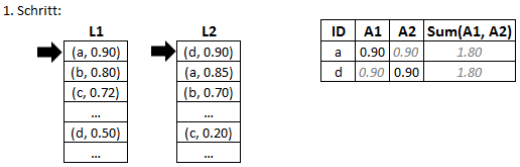

Fig. 11. Der erste Schritt des Stream Combine Verfahrens für das gegebene Beispiel.

Im nächsten Schritt werden die Listen weiter expandiert. Nach diesem Schritt das Dokument a vollständig bekannt. Es gibt kein anderes Dokument mit einem höheren Retrievalwert, also kann es ausgegeben werden (siehe Fig. 12).

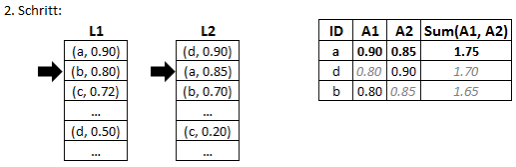

Fig. 12. Der zweite Schritt des Stream Combine Verfahrens für das gegebene Beispiel.

Im letzten Schritt werden nun erneut die Listen expandiert. Der fehlende Wert für Dokument b wird gefunden. Es gibt kein Dokument mehr, das einen höheren Retrievalwert hat als b[4], daher kann das Dokument ausgegeben werden. Es wurden nun 2 Dokumente ausgegeben und der Algorithmus terminiert (siehe Fig. 13).

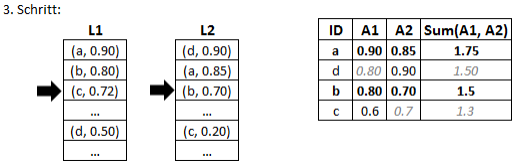

Fig. 13. Der dritte Schritt des Stream Combine Verfahrens für das gegebene Beispiel.

Der Stream Combine Algorithmus bietet einigen Spielraum für, in [3] vorgestellte, Verbesserungen. Das Ziel ist bei diesen immer den Algorithmus früher terminieren zu lassen. Da der Algorithmus erst terminiert, wenn k Dokumente ausgegeben wurden, muss es das Ziel sein, diese so schnell wie möglich auszugeben. Sie können aber erst ausgegeben werden, wenn sie vollständig bekannt sind und kein besseres Dokument mehr existiert, bei dem noch Gewichte unbekannt sind.

[4] Der Retrievalwert von d muss aufgrund des eingesetzten minimalen Wertes als obere Grenze kleiner oder gleich dem Retrievalwert von b sein.

Aus diesem Grund wäre es beispielsweise vorteilhaft, nicht alle Listen gleichzeitig zu expandieren, sondern gezielt Listen zu expandieren, deren Werte sehr schnell abfallen oder deren Werte am meisten zum Retrievalwert beitragen. Dadurch werden die oberen Grenzen für diese Liste schnell sehr klein und die Relevanzwerte der Dokumente mit unbekannten Eigenschaften fällt schneller ab, wodurch Dokumente bei denen alle Eigenschaften bekannt sind, schneller ausgegeben werden können. Bei der gezielten Expansion von Listen können Heuristiken genutzt werden, die die beste Liste bestimmen.
Ein anderer Ansatz ist es, statt den sortierten Top-K Dokumenten lediglich eine Menge der Top-K Dokumente zu fordern. Sobald ein Dokument in allen Listen gefunden wurde, also vollständig bekannt ist, können wir uns sicher sein, dass es unter den Top-K Dokumenten ist. Lediglich die genaue Position ist noch unbekannt, falls es noch Dokumente mit einem besseren Retrievalwert gibt. Ist also nur die Menge der Top-K Dokumente gefordert, kann das komplett bekannte Dokument bereits ausgegeben werden und so der Algorithmus früher terminieren.

6 Fazit

In dieser Ausarbeitung wurde eine Reihe von Algorithmen zur Bestimmung der Top-K Dokumente in einem IR-System vorgestellt. Ihnen allen gemein ist die prinzipielle Vorgehensweise. Sie versuchen möglichst früh zu terminieren und somit nicht über die kompletten Listen, die für einen Term existieren, zu iterieren. Damit sie dies können, nutzen sie alle die Sortierung der Listen, sowie die Monotonie der Retrievalfunktion aus. Ein kritisches Thema dabei sind Random Accesses, da diese sowohl das Verfahren beschleunigen, als auch verlangsamen können. Ein anderer interessanter Punkt ist die Wahl einer Liste zum Expandieren. In der Ausarbeitung wurden die Listen immer parallel durchlaufen. Jedoch bietet dieses Vorgehen Optimierungspotential. Insbesondere bei Verfahren wie dem Threshold-Algorithmus macht es Sinn zuerst Listen zu expandieren, bei denen die Werte sehr schnell sehr klein werden. Dadurch kann der Grenzwert schnell kleiner werden und der Algorithmus früher terminieren. Um herauszufinden welche Listen expandiert werden sollten, können Heuristiken eingesetzt werden.
Ein anderer Ansatz sind sogenannte Approximation Algorithms (siehe [3]). Diese bestimmen nicht die genauen Top-K Dokumente, sondern lediglich die wahrscheinlichen Top-K Dokumente. Sie nutzen dabei die Tatsache, dass der Nutzer meistens nicht an den wirklichen Top-K interessiert ist, sondern sehr oft einfach an relevanten Dokumenten interessiert ist. Aus diesem Grund können die hier vorgestellten Algorithmen für die Praxis zu streng definiert sein, wenn sie fordern, dass wirklich die k Dokumente mit dem besten Retrievalwert zurückgegeben werden sollen (siehe [3]).

7 Literaturverzeichnis

1. Fagin, Ronald and Lotem, Amnon and Naor, Moni. Optimal aggregation algorithms for middleware. Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems: 102-113, 2001.
2. Ulrich Güntzer and Wolf-Tilo Balke and Werner Kießling. Towards Efficient Multi-Feature Queries in Heterogenous Environments. Proceedings of the IEEE International Conference on Information Technology: Coding and Computing, Las Vegas: 622-628, 2001.
3. Ralf Schenkel. Effizienzaspekte von Information Retrieval. Präsentation bei der Herbstschule der GI-Fachgruppe Information Retrieval, Schloss Dagstuhl, Oktober 1, 2008.
4. Shmueli-Scheuer, Michal and Li, Chen and Mass, Yosi and Roitman, Haggai and Schenkel, Ralf and Weikum, Gerhard. Best Effort Top-K Query Processing Under Budgetary Constraints. 25th IEEE International Conference on Data Engineering, ICDE 2009 : 928-939, 2009.
5. Thomas Gottron. Web Information Retrieval. Präsentation bei der Summer Academy der Universität Koblenz-Landau, Koblenz, 2011.