Ausarbeitung im Rahmen des Seminars
"Peer-To-Peer Information Systems“ (WS03/04)
zum Thema

ODISSEA:

A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval

Torsten Suel, Chandan Mathur, Jo-Wen Wu, Jiangong Zhang, Alex Delis, Mehdi Kharrazi, Xiaohui Long, Kulesh Shanmugasundaram

Referent: Daniel Porta
Betreuer: Christian Zimmer

Universität des Saarlandes, Saarbrücken
Max-Planck-Institut für Informatik
AG5: Databases and Information Systems Group
Prof. Dr.-Ing. G. Weikum

INHALT

1. Motivation

Möchte man heutzutage im Internet etwas recherchieren, so benutzt man üblicherweise eine der großen crawling-basierten Suchmaschinen wie Google oder Altavista. Gleichermaßen beliebt wie benutzt sind gegenwärtig auch einschlägig bekannte Peer-To-Peer Systeme, die vornehmlich zum File-Sharing benutzt werden. Aufgrund der großen Verbreitung solcher Systeme ist es erforderlich geworden in diesen Netzwerken gezielt nach Informationen suchen zu können. Leider bereitet die dezentrale Organisation von Gnutella und Co. den Entwicklern von P2P Suchmaschinen in manchen zentralen Designfragen einige Kopfschmerzen. Dabei ist das Suchen in P2P-basierten Tauschbörsen vergleichsweise einfach zu bewerkstelligen, weil eine Vielzahl von Date(ie)n redundant im System abgelegt ist. Wie sucht man aber nach einer bestimmten Datei, die genau einmal im System gespeichert ist? Dazu benötigt man sehr viel Rechenkapazität, die nicht gerade im Überfluss vorhanden ist. Es drängt sich die Frage auf: woher nehmen wenn nicht stehlen? Die einzelnen Peers in einer P2P Umgebung stellen gemeinsam genügend freie Rechen- und Speicherkapazität zur Verfügung um ein solches Vorhaben zu realisieren. Begünstigend kommt hinzu, dass moderne Computer immer leistungsfähiger werden und sich die Netzwerkbandbreite, vor allem im Internet, ständig vergrößert.

Das im Folgenden vorgestellte System ODISSEA [1] (open distributed search engine architecture) greift obige Thematik auf und versucht auftretende Hindernisse, die die P2P Suche mit sich bringt, clever zu lösen. Dazu bedienen sich die Entwickler von ODISSEA einiger interessanter Ansätze und Ideen, die auf den ersten Blick etwas unkonventionell erscheinen mögen, aber durchaus ihre Berechtigung haben.

2. System Design Details

In der Hauptsache unterscheidet sich ODISSEA von anderen P2P Suchmaschinen durch eine 2-schichtige Architektur, dem sog. Two-Tier, bzw. Two-Layer Approach und der Verwendung eines globalen, anstelle eines lokalen Index.

2.1 Two-Layer Approach

Die Grundlage von ODISSEA bildet die P2P-basierte untere Schicht. Diese zeichnet sich verantwortlich für die Verwaltung der teilnehmende Peers oder Knoten, d.h. sie registriert neue Knoten und entfernt solche aus den Registern, die das System verlassen haben. Weiterhin obliegt es der unteren Schicht neu eingefügte oder aktualisierte Dokumente zu speichern und auf diese im globalen Index zu verweisen und gestellte Suchanfragen auszuführen.

Die obere Schicht bilden sog. Clients. Clients sind nicht P2P-basiert, sondern interagieren lediglich mit der unteren Schicht. Es gibt zwei Arten von Clients, welche speziell nach den Wünschen des Benutzers implementiert werden können:

1. **Update Clients** sind beispielsweise Crawler oder Webserver. Crawler durchforsten entweder ODISSEA selbst oder aber das Internet und fügen die gecrawlten Inhalte in ODISSEA ein. Alternativ können Webserver ihren Inhalt in ODISSEA einfügen.
2. **Search Clients** stellen der unteren Schicht Suchanfragen und bestimmen deren Ausführung durch sog. „query execution plans“. Das daraufhin gelieferte Suchresultat wird vom Search Client ausgewertet und nach Relevanz sortiert.

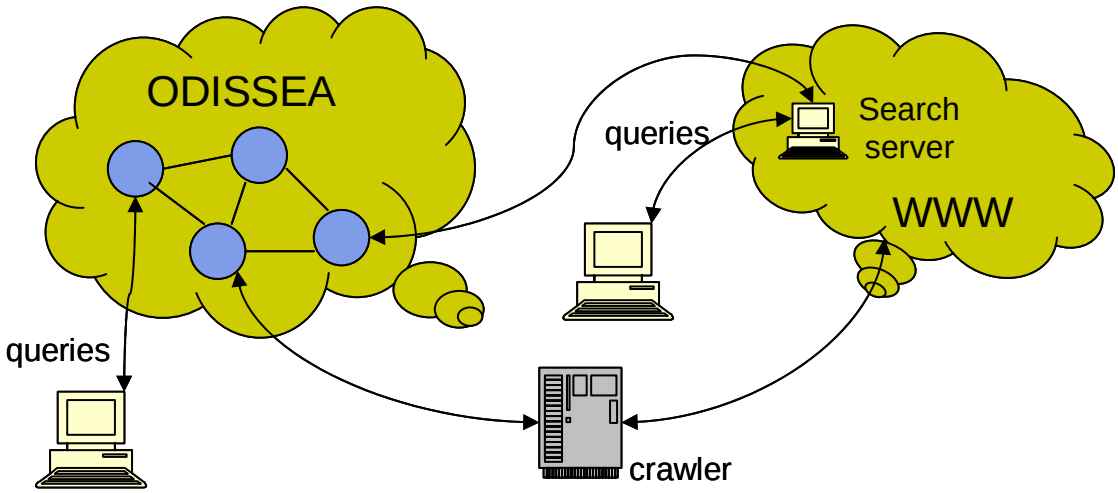

Abbildung 1 - Schematische Darstellung der ODISSEA Architektur

Eine solche Aufteilung erreicht eine große Vielfalt an unterschiedlichen Suchwerkzeugen, die die Rechenkapazität auf Clientseite besser ausnutzen und deren einzige Gemeinsamkeit darin besteht, dieselbe untere Schicht zu benutzten. Damit solche Tools entwickelt werden können, wird eine API bereitgestellt, die im Vergleich zur Google-API als eher low-level zu betrachten ist. Das unterstützt die gewollte Vielfalt, hat aber leider auch zur Folge, dass die Entwickler mehr auf Details achten müssen, soll ihr Client zu künftigen Suchanfragen schnell Ergebnisse liefern. Eine andere Gefahrenquelle besteht darin, dass im schlimmsten Fall sehr viele Daten zu einem Search Client transferiert werden muss, falls auf Serverseite von ODISSEA keine Vorevaluation der Resultate durchgeführt wird und das Ranking komplett beim Client stattfindet. Deshalb ist an dieser Stelle wohl ein Mittelweg einzuschlagen, der schonender mit der zur Verfügung stehenden Bandbreite umgeht.

2.2 Globaler Index

Wie bereits erwähnt verwendet ODISSEA anstelle eines lokalen einen globalen Index. Um die Unterschiede aufzuzeigen, führen wir zunächst ein paar Begriffe ein.

Ein *Posting* zu einen Term t und einem Dokument d ist ein Tupel bestehend aus der DokumentID von d, der Position von t innerhalb von d und weiteren zusätzlichen Angaben die die Gewichtung von d bzgl. einer Suchanfrage mit der Query t beeinflussen können. Eine solche Angabe wäre z.B., ob t in d fett, unterstrichen oder besonders groß dargestellt ist. Ein typisches Posting zu einem Term t hat die Form: <*DocID, Position, zusätzliche Angaben*>.

Eine *invertierte Liste* zu einem Term t und einer Kollektion von Dokumenten D ist eine Liste von *Postings*, die alle Auftreten von t in D repräsentiert.

Ein *invertierter Index* ist eine Menge von Termen mit ihren dazugehörigen *invertierten Listen*.

Ein *lokaler (invertierter) Index* beinhaltet alle Objekte (Terme und Dokumente), die lokal auf dem Peer abgelegt sind. Ein *globaler (invertierter) Index* zu einer Menge

von Termen auf einem bestimmten Knoten umfasst alle Auftreten der Terme im gesamten P2P-Netz.

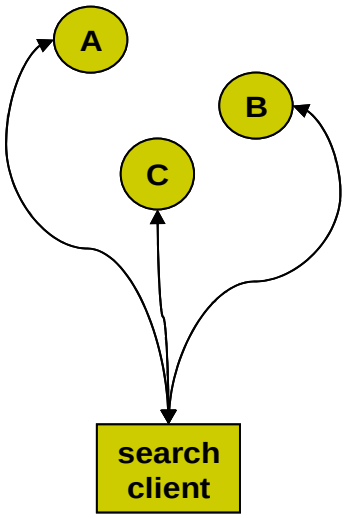

Abbildung 2 – Query-Ausführung bei lokaler Index-Organisation

Nehmen wir an, wir würden eine Query q an ein System mit lokaler Index-Organisation stellen. Die naive idealisierte Ausführung von q würde dann so aussehen, dass der Search Client q an jeden Knoten im System schicken muss und dann von allen eine Antwort erwartet. Umfasst das System sehr viele Knoten oder anders ausgerückt, ist das System sehr hoch skaliert, dann ist das praktisch nicht möglich, vor allem wenn man in Betracht zieht, dass manche Knoten sehr lange für eine Antwort benötigen oder gleich gar nicht antworten. Abhilfe würde eine Zeitschranke schaffen, die angibt, wie lange der Client auf Antworten warten soll. Ein solches Vorgehen würde aber mit an Sicherheit grenzender Wahrscheinlichkeit die Qualität des Suchergebnisses negativ beeinflussen, weil die Inhalte mancher Peers überhaupt nicht berücksichtigt werden.

Deshalb hat man sich bei ODISSEA für eine globale Index-Organisation entschieden. Nehmen wir an, wir würden nach Dokumenten suchen, die die Terme „chair“ und „table“ enthalten. Naiver Weise würde der Client daraus nun zwei Anfragen generieren. Erstens, will der Client vom Knoten, dessen Index den Term „chair“ enthält die invertierte Liste zu „chair“ und zweitens will er dasselbe für „table“. Aus dem Schnitt der Listen berechnet der Client dann das endgültige Ranking. Eine Lösung mit einem intelligenteren query execution plan würde folgendermaßen aussehen: der Client verschickt die Query zuerst an den „chair“-Knoten (wir nehmen an, dass die invertierte Liste von „chair“ kleiner ist als die von „table“). Dieser sendet dann die Query mit der invertierten Liste von „chair“ an den

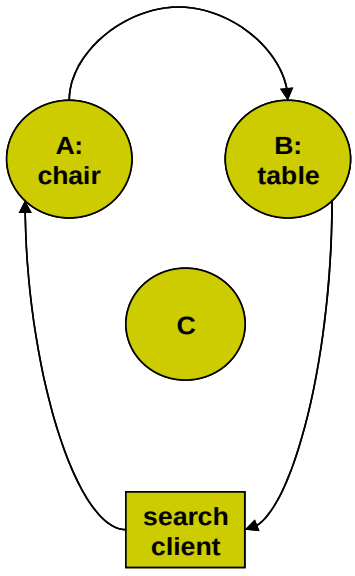

Abbildung 3 – Query-Ausführung bei globaler Index-Organisation

„table“-Knoten. Dort wird der Schnitt der beiden Listen gebildet, der dann an den Client zurück gesendet wird.

In beiden Fällen gibt es jedoch ein generelles Problem: das zu versendende Datenvolumen ist zu groß. Das betrifft zum einen den initialen Aufbau des Index, wo Millionen Postings nahezu gleichzeitig versendet werden müssen (zum Aufbau eines lokalen Index ist keine Netzwerk-Kommunikation erforderlich), zum anderen können solche invertierte Listen auch mal schnell 100MB groß sein. Ohne entsprechende Pruning-Strategien, die wir im übrigen später kennen lernen werden, wäre die Response Time, das ist die Zeit die man auf ein Suchresultat wartet, viel zu groß.

Allgemein entscheidet sich die Frage nach der Art der Index-Organisation dadurch, wie oft der Index aktualisiert wird, und damit wie viele Postings ständig verschickt werden, die die restliche Bandbreite schmälern, und wie dynamisch das System ist. Ist das System zu dynamisch, d.h. kommen viele Knoten hinzu und fallen viele weg, dann ergeben sich weitere Schwierigkeiten in Form von erhöhtem Verwaltungsaufwand für das Einfügen oder Aktualisieren von Dokumenten oder das Erstellen von Sicherheitskopien, sog. Replikate, die erforderlich sind, sollte ein Knoten ausfallen. Um dieses Problem zu umgehen, nehmen wir im Folgenden an, dass ODISSEA in einer relativ stabilen P2P-Umgebung im Einsatz ist.

2.3 Crawling Approach

Dadurch, dass das Crawling von den sog. Crawling Clients vorgenommen wird, ist es sehr einfach einen Crawler neuen Umständen anzupassen, ohne das die P2P-Umgebung davon etwas mitbekommt oder ebenfalls geändert werden muss. Einen zusätzlichen Vorteil erhält man durch den Verzicht auf P2P-basiertes Crawling. Andere, intelligentere Suchstrategien als BFS sind in einer P2P-Umgebung ohne den Einsatz eines zentralen Schedulers nicht zu implementieren.

3. Technische Details

Zurzeit arbeiten die Entwickler an einem ersten Prototyp in Java. Dazu setzten sie Pastry [2] als untere P2P Schicht ein. Pastry wurde im Rahmen des Seminars zwar

nicht behandelt, jedoch ist die Funktionsweise im weitesten Sinne ähnlich zu Chord. Pastry verfügt über einen zirkulären KnotenID-Raum, sowie über Routing- und Leaf-Node-Tables. Alle Objekte (Terme und Dokumente) werden durch einen 80-Bit-Schlüssel identifiziert. Diesen erhält man durch MD5, einem aus Unix bekannten Verschlüsselungsalgorithmus. MD5 bekommt ais Eingabe den Namen des Terms in der Form *index://term* oder die URL des Dokuments. Diese Zeichenkette hasht MD5 dann auf den 80-Bit-Schlüssel. Um zu bestimmen, auf welchem Knoten das Objekt abgelegt wird, wird ein DHT-Mapping [3] verwendet. Damit werden die Schlüssel auf eine dem System bekannte IP-Adresse gehasht.

3.1 Einfügen von Dokumenten

Wird ein neues Dokument in ODISSEA eingefügt, so wird erst mittels DHT bestimmt, auf welchem Knoten das Dokument abgelegt werden soll. Dort wird es gespeichert, und ein Parser, der lokal auf dem Knoten läuft, generiert zu jedem Term im Dokument ein entsprechendes Posting. Diese Posting werden nun an die Knoten mit den entsprechenden Index-Strukturen gesendet. Dies geschieht jedoch nicht, indem man für jedes Posting eine eigene TCP-Verbindung aufbaut, sondern vielmehr durch das in Pastry festgelegte Routing. Am Zielknoten angelangt, wird ein Posting dann in der richtigen invertierten Liste im Index gespeichert. Lokal, d.h. auf einem Knoten, ist ein Index in zwei Index-Strukturen aufgeteilt. Eine große ist auf der Festplatte abgelegt und eine kleine im Hauptspeicher. Ankommende Postings werden zunächst in die kleinere Struktur eingefügt, die ab und zu, sollte sie zu groß für das RAM sein, mit dem großen Index verschmolzen wird. Diese Organisation ermöglicht geringe amortisierte Zeitkosten für eine Einfüge-Operation, weil langsame Festplattenzugriffe vermieden werden.

3.2 Gruppieren und Teilen

Wie schon erwähnt besitzt jeder Knoten seinen eigenen Index. Eigentlich sind es zwei, da Terme und Dokumente getrennt gespeichert werden. Dokumente werden im Dokument-Index gespeichert und Postings im Term-Index. Ein Index besteht im Wesentlichen aus einer Berkeley Datenbank [4], die frei erhältlich ist und viele Programmiersprachen mit einer entsprechenden API unterstützt. Ein

Hauptbestandteil der Datenbank ist ein sog. B+ Baum. Ein B+ Baum besitzt einen hohen Fanout und ist somit sehr flach, was ein effizientes Suchen, Einfügen und Löschen ermöglicht. Die eigentlichen Informationen befinden sich an den Blättern, die inneren Knoten erledigen nur das Routing zu den Blättern. Damit eine solche Struktur erst Sinn macht, müssen mehrer Objekte zusammen gruppiert werden (für einen einzigen Term benötigt man ja keinen B+ Baum). Das geschieht so:

Man fasst alle Objekte zu einer Gruppe zusammen, deren IDs auf den ersten w Bits übereinstimmen. Gleichzeitig stellen diese w Bits auch die GruppenID dar, die bestimmt, auf welchem Knoten (mittels DHT) die Gruppe gespeichert wird. In unserem Fall ist w auf 16 festgelegt. Die restlichen Bits einer ID werden vorläufig noch nicht betrachtet. Diese bekommen erst Bedeutung, falls eine Gruppe zu groß wird. Wird ein Gruppen-Index für Dokumente größer als 1GB, wird die Gruppe anhand des w+1 Bits in zwei neue Gruppen geteilt. Da die neuen Gruppen nun eine neue, längere ID besitzen, werden sie dann normalerweise auch auf anderen Knoten abgelegt. Damit die Informationen danach noch verfügbar sind, verbleibt an alter Stelle eine sog. stub structure, die einzig und allein dafür verantwortlich ist, Anfragen oder neue Dokumente an die entsprechende, neue Speicherstelle mittels des w+1 Bits weiterzuleiten. Wird eine invertierte Liste innerhalb einer Index-Struktur für Terme größer als 100MB, so wird diese Liste anhand der DokumentIDs der einzelnen Postings geteilt. Diese Listen werden dann ebenfalls auf anderen Knoten abgelegt, und zurück bleibt eine stub structure.

3.3 Replikation und Synchronisation

Replikation wird auf Gruppenebene durchgeführt, damit im Falle von fehlerhaften, zurzeit unerreichbaren oder verlassenen Knoten sichergestellt ist, dass keine Informationen verloren gehen. Die Replikate einer Gruppe werden einfach durch anhängen von „/0“, „/1“, usw. an die GruppenID durchnummeriert (z.B. 01001001011/2) und formen zusammen eine sog. Clique. Das verkompliziert die Sache natürlich etwas, weil nun die erweiterte ID während Lookups im System benutzt wird. Aufwändiger jedoch ist das Einfügen von neuen Objekten. Um die Replikate konsistent zu halten, wird ein Objekt zuerst in ein beliebiges Replikat eingefügt, welches dann dafür verantwortlich ist, die anderen Mitglieder der Clique zu

synchronisieren. Dies sollte innerhalb von wenigen Minuten geschehen, weil sonst evtl. veraltete Suchresultate geliefert werden. Ein Knoten kann auch Replikate mehrerer Gruppen beherbergen. Dann ist ein Knoten teil mehrerer Cliquen. Eine Clique ist dafür verantwortlich, dass immer genügend Mitglieder erreichbar sind. Dazu kommunizieren die Replikate einer Gruppe gelegentlich untereinander, um den Status der anderen Mitglieder festzustellen.

Wird ein Ausfall entdeckt, so wird der Schaden durch die Clique behoben, sprich ein neues Replikat erstellt. Um die Bandbreite zu schonen wird jedoch erst ein neues Replikat erstellt, wenn eine kritische Situation erreicht ist. Eine Situation ist erst dann kritisch, wenn eine gewisse Anzahl an Replikaten zurzeit nicht erreichbar ist oder ein Replikat das System endgültig verlassen hat und nicht mehr mit einer Rückkehr zu rechen ist. Letzteres wird durch die Zeit bestimmt, die ein Knoten temporär nicht erreichbar war. Kommt ein Knoten zurück ins System, muss er mit den anderen Replikaten synchronisiert werden, um die Konsistenz zu wahren. Dies geschieht momentan noch anhand von Logfiles die die verpassten Index-Updates (mit einem Zeitindex versehen) abspeichern.

4. Effiziente Ausführung von Suchanfragen

4.1 Hintergrund

Sein d ein Dokument, q = $q_0 \ldots q_{m-1}$ ein Query, das aus m Suchtermen besteht und sei F eine Funktion, welche d in Abhängigkeit von q einen Wert F(d,q) zuweist. Eine solche Funktion heißt *Ranking-Funktion*.

Eine übliche Form einer Ranking-Funktion sieht wie folgt aus:

$$F(d,q) = \sum_{i=0}^{m-1} f(d,q_i)$$

Dabei erhält man F(d,q) durch aufaddieren der Teilgewichtungen f(d,qi) aller q_i in d. Beispielsweise kann f eine boolesche Funktion sein, die wahr liefert, wenn q_i in d vorkommt und ansonsten falsch. Eine andere Möglichkeit für f wäre die bekannte TF*IDF-Formel.

Oft genügt es, zu einer Query nur die k besten Dokumente, d.h. Dokumente mit dem höchsten F-Wert, zurückzuliefern. Dieses Vorgehen bezeichnet man als sog. *Top-k Ranking*.

Durch Auswerten von Logfiles hat man festgestellt, dass Queries typischerweise höchstens zwei Suchterme enthalten. Der folgende Algorithmus konzentriert sich deshalb auf das Bestimmen der k besten Dokumente zu einer Query, die aus genau zwei Termen besteht – und zwar ohne den kompletten Schnitt der beiden zugehörigen invertierten Listen zu berechnen.

4.1 Fagin's Algorithmus

Angenommen unser Query hätte die Form $q = q_0$ AND q_1. Dann ist die Wahrscheinlichkeit, dass ein Dokument, das sich unter den Top-k befindet, in mindestens einer der beiden Unterkategorien sehr hoch bewertet wird. Weiterhin gehen wir davon aus, dass die Postings der invertierten Listen die Form $(d, f(d,q_i))$ haben und nach der zweiten Komponente absteigend sortiert sind. Außerdem ist es für die Ausführung des Algorithmus wichtig, dass sich die invertierten Listen zu q_0 und q_1 lokal auf demselben Rechner abgelegt sind. Das Ziel besteht nun darin, die Top-k Dokumente zu q so schnell wie möglich zu bestimmen. Das funktioniert so:

1. Lese jeweils gleichzeitig ein Element der beiden Listen, beginnend vom ersten. Vergleiche nach jedem gelesenen Paar, ob die Präfixe Dokumente enthalten, die in beiden vorkommen. Mache das solange, bis k Übereinstimmungen gefunden wurden. (hier k=2)
2. Berechne für diese k Dokumente das endgültige Ranking. Für jedes Dokument, zu dem keine Übereinstimmung in einem Präfix gefunden wurde, führe ein Lookup in der jeweils anderen Liste durch, um eine Gewichtung zu erhalten. Dokumente, die danach immer noch „alleine" sind werden im Ranking nicht berücksichtigt, da sie das Query nicht erfüllen.
3. Gib die Top-k Dokumente mit der höchsten Gewichtung aus (hier d_1, d_5)

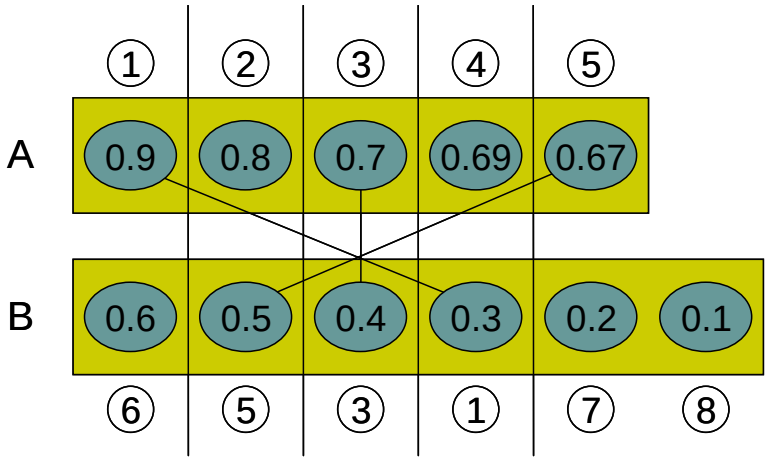

Abbildung 4 - Fagin's Algorithmus am Beispiel

Der Algorithmus arbeitet korrekt. Angenommen der erste Schritt ist beendet, der Algorithmus hat k Dokumente gefunden und direkt hinter der Barriere befindet sich ein weiteres Dokument, das in beiden Listen vorkommt. Dann kann dieses Dokument nicht zu den Top-k gehören, weil die Einzelgewichtungen in den Listen A und B jeweils kleiner sind als die des k-ten gefundenen Dokuments. Für alle Dokumente, die eine größere und (evtl.) eine kleine Einzelgewichtung haben als das k-te gefundene Dokument, wird ein Lookup durchgeführt.

4.3 Threshold Algorithmus

- Durchlaufe beide Listen gleichzeitig und lese $(d,f(d,q_0))$ von der ersten und $(d',f(d',q_1))$ von der zweiten Liste
- Berechne $t = f(d,q_0) + f(d',q_1)$
- Für jedes Dokument in eine der Listen führe ein Lookup in der anderen durch, um die endgültige Gewichtung zu bestimmen
- Der Algorithmus terminiert, sobald k Dokumente gefunden wurden, deren Gewichtung höher ist als der momentane Wert von t

Weil es keinen Sinn macht, die beiden Listen gleichzeitig zu durchlaufen während sie in einem P2P Netzwerk verteilt sind, müssen die gezeigten Techniken angepasst werden. Das führt uns zum nächsten Protokoll, das darauf ausgelegt ist so wenig Daten wie möglich über das Netz zu versenden, um die Bandbreite zu schonen und um schnell zu sein.

4.4 Simple Distributed Pruning Protocol (DPP)

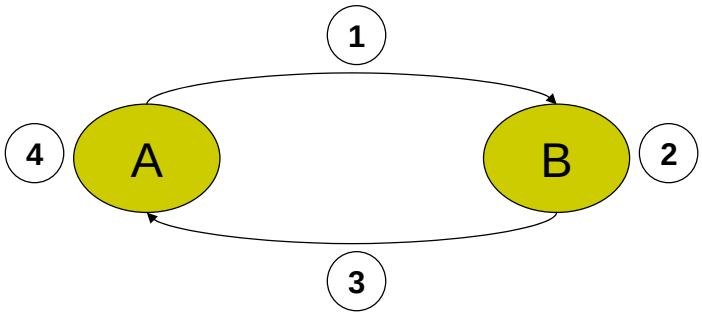

Abbildung 5 - Schematische Darstellung des DPP

1. Knoten A sendet die ersten x Postings zum Knoten B. Sei r_{min} der kleinste Wert $f(d,q_0)$, der versendet wird. Nun stellt sich sicherlich die Frage, woher kommt das x und wie groß muss es sein. Dazu gleich mehr.
2. Knoten B empfängt das Präfix und führt für jedes Posting ein Lookup durch, um die endgültige Gewichtung zu bestimmen. Behalte die k Dokumente mit der höchsten Gewichtung. Sei r_k der kleinste Wert $f(d,q_0) + f(d,q_1)$. Nun erkennt man, wie die Beschaffenheit von x sein muss. Auf der einen Seite darf x nicht zu klein sein, weil B ansonsten womöglich keine k Dokumente findet und ein weiterer Durchlauf erforderlich wird. Ist der Wert auf der anderen Seite zu groß, so werden zu viele Daten über das Netz transferiert.
3. Knoten B sendet nun die k-besten Dokumente wieder zurück zu A. Außerdem werden alle Postings unter den ersten x Postings in der Liste von B zu A gesendet für die gilt: $f(d,q_1) > r_k - r_{min}$. Das sind genau die, die das Ranking noch beeinflussen können.
4. Knoten A führt für alle von B erhaltenen Postings ein Lookup durch und berechnet für diese die endgültige Gewichtung. Dann werden die k Dokumente zurückgegeben, die die höchste Gewichtung haben.

Rechnen wir ein Beispiel, um uns die Funktionsweise des DPP zu verdeutlichen.
Sei k=2 und x=3.
Knoten A enthält Term q_0: (d_1, 0.9), (d_2, 0.8), (d_3, 0.7), (d_4, 0.69), (d_5, 0.67)
Knoten B enthält Term q_1: (d_6, 0.6), (d_5, 0.5), (d_3, 0.4), (d_1, 0.3), (d_7, 0.2), (d_8, 0.1)

1. A sendet zu B die ersten 3 Postings: (d_1, 0.9), (d_2, 0.8), (d_3, 0.7); r_{min} = 0.7

2. B berechnet: (d_1, 0.9 + 0.3) (d_2, 0.8 + ----) (d_3, 0.7 + 0.4)
   und behält (d_1, 1.2) (d_3, 1.1); r_k = 1.1
3. B sendet zu A die (vorläufigen) Top-2 Dokumente: (d_1, 1.2), (d_3, 1.1)
   und außerdem (d_6, 0.6), (d_5, 0.5), da $f(d_{6,5},q_1) > 0.4$
4. A berechnet: (d_6, 0.6+ ----), (d_5, 0.5+0.67)
   und liefert die Top-2 Dokumente (d_1, 1.2) und (d_5, 1.17) zurück

4.5 Probleme des DPP

Leider funktioniert das DPP nur mit Queries, die zwei Suchterme enthalten. Das ist recht unflexibel, da man die Suche kaum eingrenzen kann. Außerdem verursachen die Lookups lokal an den Knoten Festplattenzugriffe, da große Index-Strukturen normalerweise auf der Festplatte abgelegt sind. Diese relativ langsamen Operationen haben negativen Einfluss auf die Response Time. Die größte Frage aber ist und bleibt die Frage nach dem Wert von x, da die Leistung des DPP maßgeblich von x beeinflusst wird, wie bereits erläutert. Das x selbst wiederum ist abhängig vom dem gewünschten k und von den Termen der Query, da x für jedes Query optimalerweise ein anderes ist. Ein solches x, so die Entwickler, lässt sich bestimmen durch eine Formel, die man aus einer Vielzahl an Tests abgeleitet hat oder durch sampling-basierte Methoden, die den Wert von x schätzen.

4.6 Auswertung des DPP

Kommen wir zur Auswertung des DPP. Die untere Tabelle beruht auf den Ergebnissen, die man anhand eines selbsterstellten Kostenmodells berechnet hat. Dazu wurden einer Menge von ca. einer Million Queries 900 2-termige ausgesucht, um diese dann an eine Testkollektion von 120 Millionen selbstgecrawlten Webseiten (ca. 1,8TB) zu stellen. Dabei werden jedoch nur die eigentlichen Kommunikationskosten berücksichtigt, d.h. Additionen und Lookups lokal an den Knoten wurden vernachlässigt. Der Wert von x wurde durch Experimente mit dem Threshold-Algorithmus bestimmt. Gesucht wurden die Top-10 Dokumente unter Zuhilfenahme des Kosinus-Maßes als Ranking-Funktion. Die Einteilung erfolgt nach der Länge der kürzeren invertierten Liste. D.h. jede Query involviert 2 invertierte Listen (zu jedem Term eine). Sortiert man die 900 Queries nach der Länge der

kürzeren Liste und teilt die Liste anschließend in fünf Teile, erhält man untere Aufteilung.

	shortest 20%	*shorter 20%*	*middle 20%*	*longer 20%*	*longest 20%*
Shorter lists	10.401	63.853	222.948	666.717	3.371.176
# postings A → B	2.057	4.083	2.904	4.417	3.745
# postings B → A	1.486	4.084	2.891	4.413	3.745
Total bytes transferred	28.344	65.336	46.360	70.640	59.920
Total com time (400Kbps)	1.052	1.477	1.216	1.550	1.405
Total com time (2Mbps)	833	1.368	1.107	1.441	1.295

Abbildung 6 - Auswertung des DPP

Die unteren drei Zeilen sind naturgemäß abhängig von der Zahl der Postings, die versendet werden. Was jedoch auffällt, ist dass, die Anzahl der versendeten Postings und damit die Total Com Time unabhängig von der Länge der (kürzeren) Liste ist, was im Endeffekt soviel heißt wie: das System kann jede Query in (relativ) konstanter Zeit beantworten. Das ist ein wichtiges Ergebnis, dem auch ich unter Vorbehalt zustimme, denn leider ist, meiner Meinung nach, die Response Time unter Umständen deutlich höher, da man hier die lokalen Festplattenzugriffe „vergessen“ hat.

5. Mögliche Anwendungen

Die Entwickler können sich vier Einsatz-Szenarien für ODISSEA vorstellen:

- Volltext-Suche in großen Dokumentsammlungen, die sich in P2P-Gemeinden befinden
- Suche in großen firmeneigenen Intranet-Umgebungen
- Web Suche: eine umfangreiche API soll eine Vielfalt an client-basierten Tools ermöglichen, mit denen man im Web suchen kann. Dabei schöpfen diese Tools die Kapazitäten der Client-Rechner besser aus und entlasten somit die großen Server. Jedoch ist ein solches Szenario in naher Zukunft nicht zu erwarten, so die Entwickler.
- Search Middleware: Anstatt Dokumente einzufügen, können Update-Clients direkt Indexeinträge, also Postings dem Index hinzufügen. Sinnvollerweise

wären das Postings zu „starken“ Schlüsselwörtern, die den Inhalt eines Dokumentes am besten charakterisieren. Das könnte die Ausführung von Queries beschleunigen, da der Index an sich kleiner ausfallen würde. Jedoch liegt die Identifizierung der Schlüsselwörter in der Hand des Benutzers, was zu falscher Anwendung oder sogar Missbrauch führen kann.

6. Offene Fragen und Verbesserungen

Wie wir gesehen haben gibt es noch viel zu tun, damit das System eine ernstzunehmende Konkurrenz für die Großen wie Google oder Altavista wird. Darunter fällt zum einen die Entwicklung neuer Algorithmen für das effiziente Ausführen von Queries mit mehr als nur zwei Suchtermen und zum anderen die Entwicklung neuer Techniken in Bezug auf die Index-Synchronisation für Knoten, die längere Zeit nicht erreichbar waren. Weiterhin sollten neue Strategien das Load Balancing und die Erstellung von neuen Replikaten optimieren. Da es sich bei der aktuellen Implementierung wohl eher um einen Prototypen handelt, sollten alles in allem mehr Tests durchgeführt werden, um weitere Schwachstellen zu entdecken und gute Lösungen zu untermauern. Und damit mittelfristig P2P-basierte Systeme aus den Kinderschuhen der Tauschbörsen entwachsen und in spannenderen Gebieten wie die angesprochene Websuche eine neue Heimat finden.

7. Quellen

[1] ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval. With C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X Long, and K. Shanmugasundaram. 6th International Workshop on the Web and Databases (WebDB), June 2003
URL: http://citeseer.nj.nec.com/579163.html

[2] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.
URL: http://research.microsoft.com/~antr/Pastry/

[3] DHT-Mapping: Natalia Kozlova's Ausarbeitung ihres Seminarvortrags vom 16.12.03
URL: http://www.mpi-sb.mpg.de/units/ag5/teaching/ws03_04/p2p-data/12-16-writeup1.pdf

[4] BerkeleyDB: URL: http://freshmeat.net/projects/berkeleydb/?topic_id=67, http://www.sleepycat.com/