Top-k Aggregation Queries in Large-Scale Distributed Systems

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Sebastian Michel

Max-Planck-Institut für Informatik

Saarbrücken
2007

Dekan der Naturwissenschaftlich-Technischen Fakultät I	Prof. Dr.-Ing. Thorsten Herfet
Vorsitzender der Prüfungskommission	Prof. Dr.-Ing. Thorsten Herfet
Berichterstatter	Prof. Dr.-Ing. Gerhard Weikum
Berichterstatter	Prof. Dr. Peter Triantafillou
Berichterstatter	Prof. Dr. Bernhard Seeger
Berichterstatter	Prof. Dr. Christoph Koch
Tag des Promotionskolloquiums	11.07.2007

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.

Saarbrücken, den 23.05.2007

(Unterschrift)

Kurzfassung

Top-k Anfragen spielen eine große Rolle in einer Vielzahl von Anwendungen, insbesondere im Bereich von Informationssystemen, bei denen eine kleine, sorgfältig ausgewählte Teilmenge der Ergebnisse den Benutzern präsentiert werden soll. Beispiele hierfür sind Suchmaschinen wie Google, Yahoo oder MSN.

Obwohl die Forschung in diesem Bereich in den letzten Jahren große Fortschritte gemacht hat, haben Top-k-Anfragen in verteilten Systemen, bei denen die Daten auf verschiedenen Rechnern verteilt sind, vergleichsweise wenig Aufmerksamkeit erlangt.

In dieser Arbeit beschäftigen wir uns mit der effizienten Verarbeitung eben dieser Anfragen. Die Hauptbeiträge gliedern sich wie folgt.

- Wir präsentieren KLEE, eine Familie neuartiger Top-k-Algorithmen.
- Wir entwickeln Modelle mit denen Datenverteilungen beschrieben werden können. Diese Modelle sind die Grundlage für eine Schätzung diverser Parameter, die einen großen Einfluss auf die Performanz von KLEE und anderen ähnlichen Algorithmen haben.
- Wir präsentieren GRASS, eine Familie von Algorithmen, basierend auf drei neuartigen Optimierungstechniken, mit denen die Performanz von KLEE und ähnlichen Algorithmen verbessert wird.
- Wir präsentieren probabilistische Garantien für die Ergebnisgüte.
- Wir präsentieren Minerva∞, eine neuartige verteilte Peer-to-Peer-Suchmaschine.

Abstract

Distributed top-k query processing has recently become an essential functionality in a large number of emerging application classes like Internet traffic monitoring and Peer-to-Peer Web search. This work addresses efficient algorithms for distributed top-k queries in wide-area networks where the index lists for the attribute values (or text terms) of a query are distributed across a number of data peers.

More precisely, in this thesis, we make the following distributions:

- We present the family of KLEE algorithms that are a fundamental building-block towards efficient top-k query processing in distributed systems.
- We present means to model score distributions and show how these score models can be used to reason about parameter values that play an important role in the overall performance of KLEE.
- We present GRASS, a family of novel algorithms based on three optimization techniques significantly increased overall performance of KLEE and related algorithms.
- We present probabilistic guarantees for the result quality.
- Moreover, we present Minerva∞, a distributed search engine. Minerva∞ offers a highly distributed (in both the data dimension and the computational dimension), scalable, and efficient solution toward the development of internet-scale search engines.

Zusammenfassung

Der Erfolg und das stetige Wachstum des Internets treibt die Entwicklung vieler interessanter Anwendungsgebiete voran, von der Beobachtung des Internetverkehrs und Sensor-Netzwerken bis hin zu verteilten Peer-to-Peer-Suchmaschinen. Die Beobachtung des Internetverkehrs ist unabdingbar für die Analyse der Lastcharakteristiken moderner Internet-Anwendungen wie Peer-to-Peer-Suchmaschinen, Blogs oder Nachrichten-Feeds. Da die Netzwerk-Router und andere Komponenten, die der Beobachtung dienen, stark verteilt sind, werden verteilte Aggregationsalgorithmen und sogenannte Eisberg-Anfragen benötigt, bei denen die Top-k Treffer bezüglich einer Datenaggregation berechnet werden, um die observierten Daten zu analysieren. Ähnliche Anforderungen treten auf bei der Suche nach Lastanomalien wie zum Beispiel Netzwerkeinbrüchen oder Denial-of-Service-Attacken. Sensor-Netzwerke gewinnen große Bedeutung bei der Überwachung der Umwelt, wie zum Beispiel der Wasserqualität in Flüssen oder anderen Maßen. Auch hier führt die Analyse der Daten auf natürliche Weise zu verteilten Aggregationsanfragen bei denen man üblicherweise nur an den besten k Ergebnissen interessiert ist, zum Beispiel an den Flüssen mit der höchsten Nitratbelastung. Verteilte Internetsuche, basierend auf dem Konzept der Peer-to-Peer-Systeme, ist eine neu entstehende Alternative zu zentralisieren Suchmaschinen. Peer-to-Peer-Internetsuche bietet eine Reihe von faszinierenden Möglichkeiten: Ein solches System kann vom "intellekuellen" Input einer großen Teilnehmerzahl profitieren, z.B. in Form von Lesezeichen (Bookmarks) oder Click-Streams der Benutzer. Darüber hinaus besitzt es eine geringere Anfälligkeit gegenüber dem absichtlichen Verbreiten sogenannter Spam-Seiten, sowie gegen die Manipulation von Seiteninhalten. Wie in herkömmlichen Internetsuchmaschinen werden sich die Benutzer üblicherweise nur für die besten k Ergebnisse interessieren. Die gemeinsame Eigenschaft dieser Anwendungen ist die Ad-hoc-Verteilung der Daten über die Knoten eines großen Netzwerkes. Diese Daten müssen erfasst und eine Aggregationsfunktion muss angewendet werden. Das Ziel ist die Berechnung der k besten Ergebnisse. In dieser Arbeit beschäftigen wir uns mit effizienten Top-k-Anfragen in verteilten Systemen.

Wir betrachten ein verteiltes System mit N Knoten, P_j, $j = 1, ..., N$, die zum Beispiel durch eine verteilte Hashtabelle oder ein sonstiges Overlay-Netzwerk verbunden sind. Datenelemente sind entweder einfache Dokumente wie z.B. Webseiten oder strukturierte Daten. Jedes Datenelement besitzt eine Menge

von Deskriptoren, Wörtern oder Attributwerten. Für jedes Datenelement gibt es einen vorberechneten Wert pro Deskriptor. Eine invertierte Indexliste für einen Deskriptor ist eine Liste der Datenelemente, die absteigend nach Wert des Deskriptors sortiert sind. Jede Indexliste ist einem Netzwerkknoten zugeordnet (oder im Fall von Datenreplikation mehreren Netzwerkknoten).

Wir präsentieren KLEE, ein neuartiges algorithmisches Rahmenwerk für verteilte Top-k-Anfragen. KLEE ist ein approximativer Algorithmus und speziell für den Einsatz in großen verteilten Systemen optimiert worden. KLEE erlaubt es dem Anfragesteller, seine eigenen Prioritäten zwischen den konkurrierenden Zielen Effizienz und Resultatgüte zu setzen. Experimente mit KLEE und verwandten Algorithmen haben gezeigt, dass KLEE beachtliche Effizienzgewinne bei vernachlässigbar kleinen Einbußen in der Resultatgüte erzielt.

Darüber hinaus betrachten wir die Optimierung verteilter Algorithmen. Wir präsentieren GRASS, eine Familie von Algorithmen, basierend auf drei Optimierungstechniken, mit denen die Performanz von KLEE und ähnlichen Algorithmen verbessert wird: (i) Wir zeigen wie Score-Schwellwerte, die von fundamentaler Bedeutung für die Performanz der Algorithmen sind, an die Charakteristiken der Eingabelisten angepasst werden können. Durch diese Adaption werden beachtliche Gewinne erzielt. (ii) Wir beschreiben eine Technik, um hierarchische Anfragepläne zu erzeugen. Zum Beispiel ist es bei einer Anfrage mit einer langen und mehreren kurzen Indexlisten ratsam, die kurzen Indexlisten zu dem Peer mit der langen Indexliste zu schicken, um dort die Anfrage auszuführen. Nur die Top-k-Dokumente werden anschließend zum Anfragesteller geschickt. Für die Auswahl des optimalen Anfrageplans benutzen wir einen Ansatz der dynamischen Programmierung, der alle möglichen Anfragepläne berücksichtigt und den günstigsten Plan hinsichtlich unseres Kostenmodells auswählt. (iii) Für sehr große Anfragen, die eine Vielzahl von Indexlisten einbeziehen, haben wir eine Sampling-Technik entwickelt, mit deren Hilfe eine Teilmenge der Indexlisten ausgewählt werden kann. Diese Teilmenge lässt sich teilweise um ein Vielfaches effizienter verarbeiten, bringt dennoch nur kleine Einbußen in der Resultatgüte. All diese Techniken beruhen auf einem ausgeklügelten Kostenmodell.

Neben den Performanzevaluierungen durch Experimente präsentieren wir probabilistische Garantien für die Ergebnisgüte der Algorithmen.

Eine alternative Architektur zu den zuvor erwähnten Algorithmen ist es, die einzelnen Indexlisten über mehrere Knoten zu verteilen. Dies ist ein erster Schritt zu einem System mit unbegrenzter Skalierbarkeit. Wir erwarten, dass die Knoten des Systems in einer autonomen Weise das Internet durchsuchen, dabei Dokumente betrachten und Scores für diese Dokumente bzgl. der enthaltenen Terme berechnen. Diese Ergebnisse werden in Indexlisten gespeichert, eine für jeden Term. Jeder Knoten verteilt nun die zuvor erzeugten Daten, indem er die $(docId, score, term) - Tupel$ über die Netzwerkknoten verteilt. Eine Möglichkeit, dies zu tun, ist, die Anwendung einer sogenannten ordnungserhaltenden Hashfunktion, die jedem Tupel einen Platz (Knoten) im Netzwerk, basierend auf den Hashwerten der Scores plus einem term-spezifischen Versatz, zuordnet. Auch wenn dies offensichtlich die Tupel über die Knoten verteilt, entstehen

dennoch Ungleichgewichte in der Auslastung der Knoten, da die Scores üblicherweise stark ungleich verteilt sind (Zipf-ähnlich). Um diesem Problem entgegen zu wirken haben wir eine Hashfunktion entwickelt, die ordnungserhaltend und zugleich Lastbalancierend ist. Für eine effiziente Anfrageverarbeitung über die Indexlisten bringt auch diese Datenverteilung ein Problem, da eine sehr große Anzahl der Knoten kontaktiert werden muss, um die Daten einer Indexliste zu lesen. Um dieses Problem zu lösen, schränken wir die Platzierung der Tupel einer Indexliste auf eine Teilmenge aller Knoten, sogenannte term-spezifische Netzwerke, ein. Die eigentliche Top-k-Anfrageausführung beruht auf dem Prinzip der Schwellwertalgorithmen, wobei die Last der Anfrageverabeitung dynamisch auf mehrere Knoten verteilt wird. Diese Architektur ist Teil von Minerva∞, einer verteilten Peer-to-Peer-Suchmaschine. Wir haben Minerva∞ implementiert und ausführliche Performanzanalysen durchgeführt.

Summary

The success and growth of the Web and the Internet is spurring the development of an ever increasing number of interesting application classes, from Internet-scale monitoring, to aggregation queries in sensor networks, and to peer-to-peer Web searching. Internet traffic monitoring is crucial for understanding the nature of modern applications' load characteristics such as P2P file sharing, news feeds, or Blogs, and uses network instrumentation at different levels and time scales. As the underlying routers and other components of the observatory infrastructure are highly distributed, analyzing the logged data often requires distributed aggregation and iceberg queries (i.e., top-k computations over aggregated traffic measures). Similar requirements arise for detecting traffic anomalies such as network intrusions or denial-of-service attacks. Sensor networks are gaining great importance for monitoring environmental data such as water quality measures in rivers or other measurements of the physical world. Here, too, evaluating the data naturally leads to distributed aggregation queries where one is often interested only in the top-k query results, e.g., the top water streams with the highest nitrate concentration. P2P Web search is an emerging alternative to centralized search engines that bear various intriguing potentials: lower susceptibility to search engine spam and manipulation, exploitation of behavior and recommendations of users and entire user communities implicit in bookmarks, query logs, and click streams, and collaborative search for advanced expert queries. In such a setting, queries would combine page scoring information from several peers that maintain different index lists. As in standard Web search, users often look only at the top-10 results. From our point of view, the common key feature of all such applications is that the data are distributed over a number of nodes at large scale and in an ad-hoc manner, and that this data must be collected, and some aggregation function be applied, with the desired goal being the identification of the k most relevant/interesting data items. We focus on efficient top-k query algorithms in distributed environments.

We consider a distributed system with N peers, P_j, $j = 1, ..., N$, that are connected, e.g., by a distributed hash table or some overlay network. Data items are either documents such as Web pages or structured data items such as movie descriptions. Each data item has associated with it a set of descriptors, text terms or attribute values, and there is a precomputed score for each pair of data item and descriptor. The inverted index list for one descriptor is the

list of data items in which the descriptor appears sorted in descending order of scores. These index lists are the distribution granularity of the distributed system. Each index list is assigned to one peer (or, if we wish to replicate it, to multiple peers).

We present KLEE, a novel algorithmic framework for distributed top-k queries, designed for high performance and flexibility. KLEE makes a strong case for approximate top-k algorithms over widely distributed data sources. It shows how great gains in efficiency can be enjoyed at low result-quality penalties. Further, KLEE affords the query-initiating peer the flexibility to trade-off result quality and expected performance and to trade-off the number of communication phases engaged during query execution versus network bandwidth performance. We have implemented KLEE and related algorithms and conducted a comprehensive performance evaluation.

Moreover, we consider the optimization of distributed top-k queries in wide-area networks: We present GRASS, an algorithmic framework that consists of three optimization techniques. (i) we introduce a technique to efficiently leverage the knowledge of the input data characteristics to tune score thresholds that are of fundamental importance. The basic KLEE method and related algorithms transform the top-k retrieval problem into range-queries where the ranges are determined using uniform score thresholds. We propose the usage of non-uniform thresholds, and present an efficient optimization algorithm to adapt the threshold to the index-lists score distribution characteristics. (ii) we show how hierarchical query plans can be generated using the aforementioned cost model to build optimal query execution plans that drastically increase the overall performance. Consider, for example, a query with one very large and several small input lists residing on different peers. It would be better to perform the top-k query at the peer with the large list, have the small peers ship their items to the large peer, and only send the final result to the query initiator. We use a dynamic programming approach that considers all possible query plans and chooses the cheapest plan w.r.t. our cost model. (iii) we introduce a sampling method to select a subset of the input data sources that still provides accurate results but can be, at the same time, more efficiently handled. We have performed experiments on real Web data that show the benefits of distributed top-k query optimization both in network resource consumption and query response time.

In addition to the experimental evaluation of the aforementioned algorithms, we have derived probabilistic guarantees for the result quality.

An architectural alternative to the computational model that underlies our algorithms is to distribute each index list over multiple peers, as a key step towards a system with unlimited scalability. We expect that nodes will autonomously crawl the web, discovering documents and computing scores of documents, with each score reflecting a document's importance with respect to terms of interest. This results in index lists, one for each term, containing rel-

evant documents and their scores for a term. In a succeeding step, each peer distributes its set of *(docId, score, term)-triplets* across the participating peers. One way of doing so is to use a standard order-preserving hash function that assigns each triplet to a node based on the score's hash-value plus a term-specific offset. While this obviously distributes the triplets over the peers it will create a load imbalance because of the skewed (Zipf-like) score distribution typically observed in real-world index lists. To overcome this problem, we have developed a more sophisticated hash function that distributes index lists over the participating peers in a load-balancing and, at the same time, order-preserving way. But even with such a hash function it is infeasible to distribute a single index list over all peers, since this would cause a gigantic communication overhead as all peers would have to be contacted in order to retrieve the required information. To overcome this problem, we restrict the placement of the *(docid, score, term)-triplets* for a particular term to a subset of all peers. These small networks, called term index networks (TIN), help to limit the number of peers contacted during retrieval. In general, TINs can form separate overlay networks, but for simplicity we model a TIN simply as a (circular) doubly-linked list. The top-k query processing proceeds in rounds, in which a coordinator peer retrieves batches of (docId, score, term)-triplets from the nodes that are part of the query-term specific TINs. We believe that our design choices are a big step towards a scalable P2P search engine. The Minerva∞ architecture has been implemented, and performance experiments have been conducted.

Contents

Chapter 1

Introduction and Problem Statement

Top-k query processing is a fundamental cornerstone of multimedia similarity search, ranked retrieval of documents from digital libraries and the Web, preference queries over product catalogs, and many other modern applications. Conceptually, a top-k query can be seen as an operator tree that evaluates (SQL or XQuery) predicates over one or more tables, performs outer joins to combine multi-table data for the same entities or performs grouping by entities (e.g., by document ids), aggregates a "goodness" measure such as frequencies or IR-style scores, and finally outputs the top-k results with regard to this aggregation. Ideally, an efficient query processor would not read the entire input (i.e., all tuples from the underlying tables) but should rather find ways of early termination when the k best results can be safely determined, using techniques like priority queues, bounds for partially computed aggregation values, pruning intermediate results, etc.

Applications are, for instance:

- P2P Web search is an emerging alternative to centralized search engines that bear various intriguing potentials: lower susceptibility to search engine spam and manipulation, exploitation of behavior and recommendations of users and entire user communities implicit in bookmarks, query logs, and click streams, and collaborative search for advanced expert queries. In such a setting, queries would combine page scoring information from several peers that maintain different index lists. As in standard Web search, users often look only at the top-10 results.

- Network monitoring over distributed logs [DEB05]. Here items are IP addresses, URLs, or file names in P2P sharing, and we would typically aggregate values like occurrence frequencies or transferred bytes.

- Sensor networks with sensors that have local storage and can be periodically polled [MFHH05]. Here items could be chemicals that contribute to

water or air pollution, possibly in combination with specific time periods (e.g., morning hour vs. evening hour).

- Mining of social communities and their behavior [DKM+06]. Here items could be specifically defined user groups, possibly in combination with geographic zones. The aggregation would consider frequencies of postings to different blogs, or "social tags" and ratings assigned to user-created content, or statistical information from query logs (e.g., frequencies of queries, query keywords, keyword pairs, etc.) or click streams (e.g., frequencies of popular URLs).

- Mining of distributed text or multimedia corpora. Here items could be documents, or document features like prominent categories, tags, or keyword combinations, or authors and organizations who contribute to different conferences and journals, digital libraries, social communities, etc. The applications would typically be interested in aggregating frequencies or scores.

- "Reality mining" based on distributed sources of RFID recordings or cellphone tracking [ACKS06], hopefully with proper privacy-preservation in place.

1.1 Problem Statement

We consider a distributed system with N peers, P_j, $j = 1, ..., N$, that are connected, e.g., by a distributed hash table or some overlay network. Data items are either documents such as Web pages or structured data items such as movie descriptions. Each data item has associated with it a set of descriptors, text terms or attribute values, and there is a precomputed score for each pair of data item and descriptor. The inverted index list for one descriptor is the list of data items in which the descriptor appears sorted in descending order of scores. These index lists are the distribution granularity of the distributed system. Each index list is assigned to one peer (or, if we wish to replicate it, to multiple peers).

Figure 1.1 shows an example of the query initiator P_0 (sometimes also called P_{init}) and four peers $P_1, ..., P_4$.

The overall goal is to efficiently find the top-k items (documents) or, in case of approximate algorithms, come as close as possible to the true top-k results. We measure the quality of the approximate result by the fraction of documents in the approximate top-k result that are also in the true top-k result, i.e. the relative recall.

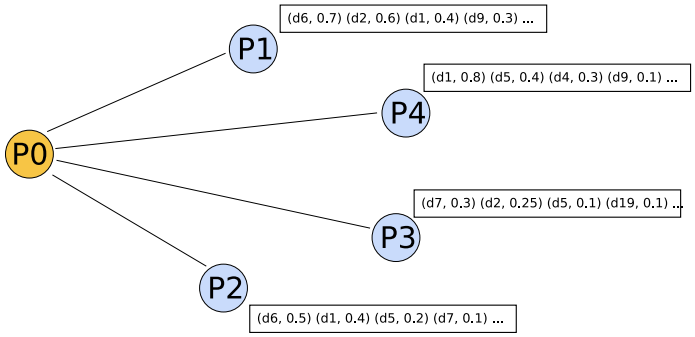

Figure 1.1: Example of a query that involves 4 data sources. $P0$ is query initiator that tries to calculate the top-k result.

1.2 Computational Model

Conceptually, the underlying data we consider resides in a (virtual) table with a schema like `Events (Id, Item, Value, ...)` with additional attributes, such as `Creator` or `Date`. The table is horizontally partitioned across many nodes in a wide-area network; partitionings are typically along the lines of value ranges, creation dates, or creators. The queries that we want to evaluate on the (virtual) union of all partitions are of the form

```
SELECT Item, Aggr(Value)
  FROM Events
  GROUP BY Item
  ORDER BY Aggr(Value)
  LIMIT k
```

We assume the following computational model. We consider a distributed system with m peers P_j, $j = 1, ..., m$. It is assumed that every node can communicate with every other node — possibly with different network costs, but without any limitation of functionality. Each peer P_j owns a fragment of the abstract relation `Events` introduced before, containing items I_j and their corresponding values $v(I_j)$. These pairs are accessible at each peer P_j in sorted order by descending value, i.e., in a (physically or virtually) sorted list L_j. These lists can be implemented by materializing local index lists, but other ways are conceivable, too. Notice that an item can, and usually does, appear in the list of more than one peer. Often, some popular items (e.g., URLs or IP addresses in a network traffic log) appear in the lists of nearly all peers.

Table 1.1 shows an example of a table holding network traffic information about users and their amount of downloaded data from a particular server on a particular day. The actual data, however, has been created in a completely decentralized way, namely on the servers where the network traffic has occurred. Transferring all access logs to a central instance would place an extremely high

Id	Server	ClientIP	Bytes	Date
1001	www.server1.com	192.168.1.4	12kB	2007/03/02
1002	www.server1.com	192.168.1.1	11kB	2007/03/06
1003	www.server2.com	192.168.1.1	7kB	2007/03/02
1004	www.server1.com	192.168.1.4	17kB	2007/03/01
1005	www.server2.com	192.168.1.1	9kB	2007/03/01

Table 1.1: Relational table containing network traffic information

Id	ClientIP	Bytes	Date
1001	192.168.1.4	12kB	2007/03/02
1002	192.168.1.1	11kB	2007/03/06
1004	192.168.1.4	17kB	2007/03/01

Id	ClientIP	Bytes	Date
1003	192.168.1.1	7kB	2007/03/02
1005	192.168.1.1	9kB	2007/03/01

Figure 1.2: Two relational tables, hosted at two different peers (servers).

burden on the central control instance that would be a bottleneck when monitoring distributed systems.

Figure 1.2 shows the data of two tables that together contain the data from Table 1.1 but now the data is distributed over two Peers. This would actually be the standard case, since HTTP access logs are generated at the place where the accesses occur. In this example, the query initiator would be interested in calculating the user (given by ClientIP) that caused the highest network traffic.

More formally: we consider queries of the form $Q = t_1, ..., t_m$, initiated at a peer P_{init}. P_{init} then aims at finding the k items with highest aggregated values over all peers P_j that hold the data for the attributes $t_1, ..., t_m$.

In Web search and other IR applications, for example, weighted summation of relevance scores for different keywords, is common practice.

In this work we consider only monotone aggregation functions, i.e. functions f that have the following property.

Definition Given two items $v = (v_1, v_2, ..., v_m)$ and $w = (w_1, w_2, ..., w_m)$ where v_i and w_j are the particular values of v and w w.r.t. the attributes $t_1, ..., t_m$. An aggregation function f is called monotone if and only if $\forall_i v_i \leq w_i$ implies that $f(v) \leq f(w)$.

For the sake of concreteness, we will use summation for value aggregation throughout this work, but weighted sums and other monotone functions are supported, too.

Scanning the list L_j allows each peer P_j to retrieve and ship a certain number of its locally highest-value items. The receiving peer (e.g., P_{init}) can then

employ a threshold algorithm [FLN03, GBK00, NR99] for value aggregation and determining whether previously unseen result candidates potentially qualify for the final top-k result, or if deeper scans or further probings of unknown values are needed to safely eliminate result candidates.

1.3 Contributions

With this work we make several contributions to the area of distributed top-k aggregation queries.

- We present the KLEE algorithmic framework as a fundamental building block towards efficient top-k query processing in distributed systems.
- We present techniques to model value distributions and show how these models can be used to reason about parameter values that play an important role in the overall performance of KLEE.
- We present the GRASS algorithmic framework. The GRASS algorithms come with three different kinds of optimization techniques: first, we introduce a technique to efficiently leverage the knowledge of the input data characteristics to tune thresholds that are of fundamental importance. Second, we show how hierarchical query plans can be generated using the aforementioned cost model to build optimal query execution plans that drastically increase the overall performance. Third, we introduce a method to select a sample of input data sources that still provides reasonably accurate results but can be, at the same time, more efficiently handled. All these techniques result in a significantly increased overall performance.
- We present probabilistic guarantees for the aforementioned algorithms.
- Moreover, we address the issue of building a highly scalable search engine. We have developed Minerva∞, a scalable and efficient Peer-to-Peer Web search engine. The distinguishing feature of Minerva∞ is the high distribution both in the data and computational dimensions. The key idea is to give up the nodes' autonomy and distribute each index list over multiple peers, as a key step towards a system with unlimited scalability. We expect that nodes will autonomously crawl the web, discovering *documents* and computing *scores* of documents, with each score reflecting a document's importance with respect to *terms* of interest.

1.4 Selected Publications

Various aspects of this thesis have been published in [MTW05a, NM07, MTW05b, BMT+05b, BMT+05a, MBTW06, MBN+06, PMW07, BMTW06].

The most important publications are the following:

Top-K Query Processing

In [MTW05a] we have presented KLEE that is also a major part of this thesis (cf. Chapter4). KLEE makes a strong case for approximate algorithms and we show that significant performance benefits can be enjoyed while having only small penalties in accuracy.

- Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. KLEE: A framework for distributed top-k query algorithms. 31st conference on Very Large Data Bases (VLDB), 2005.

In [NM07], we have addressed algebraic optimization of distributed top-k queries.

- Thomas Neumann and Sebastian Michel Algebraic query optimization for distributed top-k queries. 12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW), 2007.

P2P Web Search with Minerva and Minerva∞

We have developed Minerva [BMT+05b] and Minerva∞ [MTW05b], two Peer-to-Peer Web search engines. Minerva∞ is a major part of this thesis and is presented in Chapter 8.

- Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Minerva∞: A scalable efficient peer-to-peer search engine. ACM/IFIP/USENIX 6th International Middleware Conference, 2005.

Minerva is our fully operational Peer-to-Peer Web search prototype. Section 2.3 discusses the general idea and the architecture. In opposite to Minerva∞, Minerva makes a strong case for peer autonomy. Peers in Minerva maintain their data locally and publish only small descriptions of their local collections to a decentralized directory. These summary information are used at query time to find the most promising peers for a particular query.

- Matthias Bender, Sebastian Michel, Peter Triantafillou, Gerhard Weikum, and Christian Zimmer. Minerva: Collaborative p2p search. 31st conference on Very Large Data Bases (VLDB), 2005. Demo Paper.

1.5 Outline of this Thesis

This thesis is organized as follows. Chapter 2 presents an introduction to information retrieval and distributed systems. Chapter 3 gives an overview about existing work in the area of top-k aggregation queries. Chapter 4 presents the

KLEE algorithms, a novel family of approximate top-k algorithms. Chapter 5 presents statistical estimators and value-distribution models to reason about parameter values that are common in algorithms similar to KLEE. Subsequently, Chapter 6 addresses issues that arise when dealing with queries over many sources. It presents the GRASS algorithms that use three optimization techniques that are designed to decrease overall query response time and lower the network resource consumption. Chapter 7 presents probabilistic guarantees for the result quality of the presented algorithms. Chapter 8 presents Minerva∞, our contribution towards an efficient and highly scalable distributed search engine, where we address data placement, network organization, and query processing. Chapter 9 presents the conclusion and an outlook to future work. Appendix A contains the benchmark queries that we use to evaluate the presented approaches.

Chapter 2

Background

2.1 Introduction to Information Retrieval

Information Retrieval (IR) systems keep large amounts of unstructured or weakly structured data, such as text documents or HTML pages, and offer search functionalities for delivering documents relevant to a query. Typical examples of IR systems include web search engines or digital libraries; recently, relational database systems have been integrating IR functionality as well.

The search functionality is typically accomplished by introducing measures of similarity between the query and the documents. For text-based IR with keyword queries, the similarity function typically takes into account the number of occurrences and relative positions of each query term in a document.

Inverted Index Lists

The concept of inverted index lists [ZM06] has been developed in order to efficiently identify those documents in the dataset that contain a specific query term. For this purpose, all terms that appear in the collection form a tree-based index structure (often a B^+-tree or a trie) where the leaves contain, for each term, a list of unique document identifiers for all documents that contain this term. Conceptually, these lists are combined by intersection or union for all query terms to find candidate documents for a specific query with multiple terms. Depending on the exact query execution strategy, the inverted index lists of document identifiers may be ordered according to the document identifiers or according to a score value to allow efficient pruning.

$TF * IDF$ Measure

The number of occurrences of a term t in a document d is called *term frequency* and typically denoted as $tf_{t,d}$. Intuitively, the significance of a document increases with the number of occurrences of a query term. The number of documents in a collection that contain a term t is called *document frequency* (df_t); the *inverse document frequency* (idf_t) is defined as the inverse of df_t. Intuitively,

the relative importance of a query term decreases as the number of documents that contain this term increases, i.e., the term offers less differentiation between the documents. In practice, these two measures may be normalized (e.g., to values between 0 and 1) and dampened (e.g., using logarithms) and smoothed (e.g, using Laplace smoothing). A typical representative of this family of $tf*idf$ formulae that calculates the weight $w_{i,f}$ of the i-th term in the j-th document is

$$w_{i,j} := \frac{tf_{i,j}}{max_t\{tf_{t,j}\}} * log(\frac{N}{df_i})$$

where N is the total number of documents in the collection.

In recent years, other relevance measures based on statistical language models and probabilistic IR have received wide attention [GF98].

2.2 Peer-to-Peer Systems

The peer-to-peer (P2P) approach facilitates the sharing of huge amounts of data in a distributed and self-organizing way. These characteristics offer enormous potential benefit for search capabilities powerful in terms of scalability, efficiency, and resilience to failures and dynamics. Additionally, a P2P search engine can potentially benefit from the intellectual input (e.g., bookmarks, query logs, click streams, etc.) of a large user community participating in the data sharing network. Finally, but perhaps even more importantly, a P2P search engine can also facilitate pluralism in informing users about Internet content, which is crucial in order to preclude the formation of information-resource monopolies and the biased visibility of content from economically powerful sources.

Web search: A conceivable, very intriguing application of P2P computing is Web search. The functionality would include search for names and simple attributes of files, but also Google-style keyword or even richer XML-oriented search capabilities. It is important to point out that Web search is not simply keyword filtering, but involves relevance assessment and ranking search results. We envision an architecture where each peer has a full-fledged search engine, with a focused crawler, an index manager, and a top-k query processor. Each peer can compile its data at its discretion, according to the user's personal interests and data production activities (e.g., publications, blogs, news gathered from different feeds, Web pages collected by a thematically focused crawl). Queries can be executed locally on the small-to-medium personalized corpus, but they can also be forwarded to other, appropriately selected, peers for additional or better search results.

For this application, the P2P paradigm has a number of potential advantages over centralized search engines with very large server farms: 1) The load per peer is orders of magnitude lower than the load per computer in a server farm, so that the P2P-based global computer could afford much richer data representations, e.g., utilizing natural-language processing, and statistical learning models, e.g., named entity recognition and relation learning. 2) The local search engine of

each peer is a natural way of personalizing search results, by learning from the user's explicit or implicit feedback given in the form of query logs, click streams, bookmarks, etc. In contrast, personalization in a centralized search engine would face the inherent problem of privacy by aggregating enormous amounts of sensitive personal data. 3) The P2P network is the natural habitat for collaborative search, leveraging the behavior and recommendations of entire user communities in a social network. A key point is that each user has full and direct control over which aspects of her behavior are shared with others, which ones are anonymized, and which ones are kept private.

Web archiving: Today, virtually all Web repositories, including digital libraries and the major Web search engines, capture only current information. But the history of the Web, its lifetime over the last 15 years and many years to come, is an even richer source of information and latent knowledge. It captures the evolution of digitally born content and also reflects the near-term history of our society, economy, and science. Web archiving is done by the Internet Archive, with a current corpus of more than 2 Petabytes and a Terabyte of daily growth, and, to a smaller extent, some national libraries in Europe. These archives have tremendous latent value for scholars, journalists, and other professional analysts who want to study sociological, political, media usage, or business trends, and for many other applications such as issues of intellectual property rights. However, they provide only very limited ways of searching timelines and snapshots of historical information. A conceivable "killer application" for P2P would be to implement comprehensive Web archiving in a completely distributed manner, utilizing the aggregated resources of millions of decentralized computers, and to provide expressive and efficient "time-travel" querying capabilities for both temporal snapshot search and the analysis of timelines of specific topics.

Sensor networks: Such networks combine small devices that measure and monitor real-world phenomena such as temperature or people in office buildings, cars on highways, water levels or pollution indicators in rivers and lakes, or the avalanche danger level on mountain slopes, to give a few prominent examples. In terms of scale, the biggest current example is probably the monitoring and real-time analysis of IP packets in Internet routers. Sensors can be stationary or mobile, or even part of mobile components that form an ad hoc network without pre-configured infrastructure; cars on highways is an example of the latter. In addition to sensors, some devices may also serve as actuators as part of feedback loops or other control purposes. Many applications of sensor networks require the aggregation of values that are reported by individual sensors, in order to monitor danger levels and other thresholds. Again, this is a perfectly decentralized setting for which a P2P-based approach seems to be the most natural method of choice.

Personal data spaces and social networks: Scientists are one class of people who often maintain extensive data on their personal computers or notebooks. But there are many other categories like journalists, marketing and financial analysts, consultants, etc., and even the "common Internet user" at least

manages significant amounts of email data. While simple email management is a common service today, a better service would actually consider also the data to which email refers and thus integrate also email attachments, file versions, and many other elements of the users' electronic desktops. A truly compelling and comprehensive service would go even further by automatically classifying and organizing all relevant data items and automating many aspects of the users' work processes. This vision is sometimes referred to as a *semantic desktop* or *personal information manager*. The most promising architectural paradigm for a comprehensive solution, with ultra-high scalability, reliability, and availability, would be the P2P-based global computer. Needless to say that strong security and privacy should be prime issues in such a setting. But compared to server-based central approaches, P2P-based overlay computers have the potential for being much less vulnerable to load bursts, attacks, and sabotage.

2.2.1 Structured Overlay Networks

All structured overlay networks are based on the principle of *resource virtualization*: they map resource identifiers like keys of data items or node addresses onto a virtual address space and then allocate virtual ids onto peers. This way the storage management and search algorithms can be implemented on top of a structured overlay network without having to know about physical network properties. The virtualization infrastructure can also take care of re-mappings when peers join or leave the network.

Structured overlay networks have been discussed in the literature in three generations:

- the first generation with basic overlays that support exact-match key lookups and a scalable virtualization infrastructure,
- the second generation with additional features regarding faster routing or fault tolerance, and
- the third generation that support also advanced operations such as range queries or string matching operations.

The first generation of structured overlay networks is mostly based on *distributed hash tables (DHTs)* and related techniques. *Chord* [SMK+01] uses hashing for mapping nodes as well as data items onto a virtual ring, and then adds a logarithmic number of routing-table entries to each peer for network efficiency. hashing provides efficient incremental re-hashing when the target domain of hash function changes, for example, when nodes fail/leave or when new nodes join the network. *Pastry* [RD01] and *Tapestry* [ZKJ01] are based on Plaxton trees: nodes are assigned random ids, and a constant number of neighbor links are created for each node based on common prefixes of their ids, effectively constituting an embedding of randomized trees in the network structure. *CAN* [RFH+01] uses a d-dimensional partitioning of the virtual id space and organizes links between neighbors according to a d-dimensional torus topology. Highly related to all these approaches is also the earlier work on *scalable distributed data*

structures (SDDSs), such as LH* [LNS96] or Snowball [VBW98], but that work did not consider the problem of heavy churn (and rather focused on scalability with regard to network growth). All of the above mentioned methods provide fast and scalable lookup of data items and localization of nodes, either in time $O(log\ n)$ or $O(n^{1/d})$ where n is the number of nodes in the network; and no peer needs to maintain routing information that requires space larger than $O(log\ n)$. Good reference points for the first generation of structured overlay networks are Chord and Pastry; their prototype software has been widely adopted in the research community. We describe both approaches in more detail below.

The second generation considered a much wider variety of network topologies including butterfly, hypercube, and various kinds of trees and tries. Moreover and more importantly, it added deeper considerations on fault tolerance, churn handling, latency issues, and interoperability among multiple, possibly heterogeneous, P2P networks. For fault tolerance, systematic replication or error-correction coding were added and woven into the overlay network itself. For example, for Chord, a simple but effective method is to replicate the data items of a node on its successor or successors in the virtual ring structure; the hash function ensures that no load imbalances are created and that failure modes of successive nodes are largely independent. For low latency of request routing, routing tables of Chord-style overlays are enhanced by nodes that exhibit a recent history of short IP round-trip times; these additional neighbor links are dynamically adjusted as the network characteristics evolve over time. Finally, for interoperability several papers proposed steps towards reference architectures and their alignment with the emerging standards for P2P infrastructure, most notably, the JXTA framework [HD05]. A good reference point for the second generation of structured overlay networks is P-Grid, discussed in more detail below.

The third generation of structured overlay networks has been aiming to provide efficient support for more versatile and complete operations on top of or as integrated part of the basic overlay infrastructure. The main motivation has been to support much richer applications beyond the classical file-sharing case, for example, database-system functionalities. An operation that has received significant attention is range queries. This is of importance not just for database systems, but for all applications that refer to time attributes, for example, Web archiving and time-travel Web search. The approaches advocated in the literature typically suggest DHT variants based on order-preserving hash functions. This goes a long way, but has limitations in reconciling load balancing with (zero-tuning) self-organization. Another class of operations that researchers have started to investigate in the context of P2P systems are string operations like prefix, suffix, and substring matching. It seems generally fair to say that this current generation of P2P data management is an ongoing endeavor, likely to see more variations and new attempts on the above and further operations in the next few years.

2.2.2 Example Chord

Chord [SMK+01] is a distributed lookup protocol for efficient localization of virtual resources. It provides the functionality of a distributed hash table (DHT) by supporting the following *lookup* operation: given a key, it maps the key onto a node. For this purpose, Chord uses hashing [KLL+97]. Hashing tends to balance load, since each node receives roughly the same number of keys. Moreover, this load balancing works even in the presence of a dynamically changing hash range, i.e., when nodes fail or leave the system or when new nodes join. Chord not only guarantees to find the node responsible for a given key, but also can do this very efficiently: in an N-node steady-state system, each node maintains information about only $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. These properties offer the potential for efficient large-scale systems. The intuitive concept behind Chord is as follows: all nodes p_i and all keys k_i are mapped onto the same cyclic ID space. In the following, we use keys and peer numbers as if the hash function had already been applied, but we do not explicitly show the hash function for simpler presentation. Every key k_i is assigned to its closest successor p_i in the ID space, i.e. every node is responsible for all keys with identifiers between the ID of its predecessor node and its own ID. For example, consider Figure 2.1. Ten nodes are distributed across the ID space. Key k_{54}, for example, is assigned to node p_{56} as its closest successor node. A naive approach of locating the peer responsible for a key is also illustrated: since every peer knows how to contact its current successor on the ID circle, a query for k_{54} initiated by peer p_8 is passed around the circle until it encounters a pair of nodes that straddle the desired identifier; the second in the pair (p_{56}) is the node that is responsible for the key. This lookup process closely resembles searching a linear list and has an expected number of $O(N)$ hops to find a target node, while only requiring $O(1)$ information about other nodes.

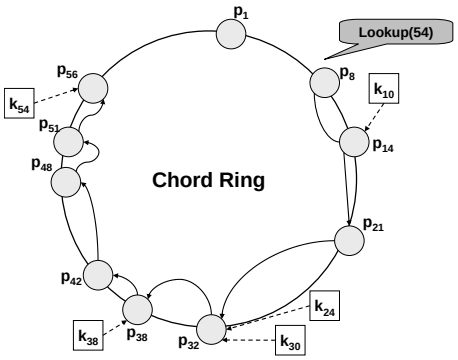

Figure 2.1: Chord Architecture

To accelerate lookups, Chord maintains additional routing information: each peer p_i maintains a routing table called *finger table*. The m-th entry in the table

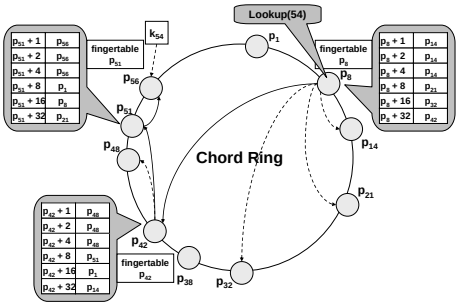

Figure 2.2: Scalable Lookups Using Finger Tables

of node p_i contains a pointer to the first node p_j that succeeds p_i by at least 2^{m-1} on the identifier circle. This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Secondly, a node's finger table does not necessarily contain enough information to *directly* determine the node responsible for an arbitrary key k_i. However, since each peer has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between itself and the target node. This property is illustrated in Figure 2.2 for node p_8. It follows that the number of nodes to be contacted (and, thus, the number of messages to be sent) to find a target node in an N-node system is $O(\log N)$.

Chord implements a stabilization protocol that each peers runs periodically in the background and which updates Chord's finger tables and successor pointers in order to ensure that lookups execute correctly as the set of participating peers changes. But even with routing information becoming stale, system performance degrades gracefully. Chord can provide lookup services for various applications, such as distributed file systems or cooperative mirroring. However, Chord by itself is not a full-fledged global storage system, and it is not a search engine either as it only supports single-term exact-match queries and does not support any form of ranking.

2.2.3 Example Pastry

Pastry is a self-organizing structured overlay network that uses a routing schema based on prefix matching. Each node is assigned a globally unique 128-bit identifier from the domain $0..2^{128}-1$, in form of sequences of digits with base 2^b where b is a configuration parameter with typical value 4. Like Chord, Pastry offers a simple routing method that efficiently determines the node that is numerically closest to a given key, i.e., which is currently responsible for maintaining that key. To enable efficient routing in an N-node network, each peer maintains a routing table that consists of $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries each, where each entry consists of a Pastry identifier and the contact information (IP

address, port) of the numerically closest node currently responsible for that key. All $2^b - 1$ entries in row n represent nodes with a Pastry identifier that shares the first n digits with the current node, but each with a different $n + 1$-st digit ($2^b - 1$ possible values). The prefix routing now works as follows: For a given key, the current node forwards the request to that node from its routing table that has the longest common prefix with the key. Intuitively, each routing hop can fix one additional digit toward the desired key. Thus, in a network of N nodes, Pastry can route a message to a currently responsible node with less than $\lceil log_{2^b} N \rceil$ message hops.

2.2.4 Example P-Grid

P-Grid [Abe01, ADH05] is a peer-to-peer lookup system based on a virtual distributed search tree. Each peer stores a partition of the overall tree. A peer's position is determined by a binary bit string (called the *path*) representing the subset of keys that the peer is responsible for. P-Grid's query routing approach is as follows: For each bit in its path, a peer stores a reference to at least one other peer that is responsible for the other side of the binary tree at that level. Thus, if a peer receives a request regarding a key it is not responsible for, it forwards the request to a peer that is "closer" to the given key. This process closely resembles the prefix-based routing approach taken by Pastry. The peer paths are not determined a priori but are acquired and changed dynamically through negotiation with other peers. In the worst case, for degenerated data key distributions, the tree shape no longer provides an upper bound for search cost as it might be up to linear depth in network size. However, it can be shown by theoretical analysis that for a (sufficiently) randomized selection of links to other peers in the routing tables, probabilistically the search cost in terms of messages remains logarithmic, independently of the length of the paths occurring in the virtual tree.

2.2.5 DHTs for Global Storage and Web Search

From the viewpoint of the overlay infrastructure, global storage can be seen as an application, layered on top of a DHT or other structured overlay network. However, it is a generic and highly versatile application that itself deserves prime attention. Several proposals have been made in the literature for building global file systems on top of a P2P overlay network.

Oceanstore (actually, its prototype implementation coined Pond) [KBC+00] is built on top of Tapestry. It virtualizes file ids (or file names) and assigns them to network nodes in a randomized manner. For efficient lookup, Plaxton trees are the mechanism that Tapestry provides in the overlay infrastructure. As an additional lookup accelerator, Oceanstore gives each node a staged set of Bloom filters, one filter for each distance level, for efficient probabilistic location of files that reside at topologically nearby nodes. For fault tolerance, error-correcting code (ECC) blocks are computed and stored at separate nodes; more specifically, Reed-Solomon codes are used to this end. As the reconstruction of corrupted

file blocks is an expensive operation with ECCs alone, full-content blocks are additionally cached/replicated at additional nodes. Updates are handled by a no-overwrite versioning approach for all files, and concurrent updates are handled by a conflict resolution method that can be made application-driven by appropriate hooks into Oceanstore. For example, latest-update-wins could be a conflict-resolution policy but more sophisticated predicate-based policies are supported as well. All aspects of the conflict resolution for updates and the fault tolerance by ECCs are managed by a specifically trusted core set of nodes, the so-called "inner ring" of Oceanstore. This resembles the super-peer architecture that most commercial P2P systems have adopted for MP3 and other file sharing. Strictly speaking, these are not perfectly scalable and completely self-organizing architectures, as super-peers are different from normal peers and are assumed to be more carefully administered than the average personal computer on the Internet.

Recently, various kinds of higher-level data managers have been proposed in a P2P setting, most notably with database system and search engine functionalities. In the first line, the best examples are PIER [HCH+05], ObjectGlobe [BKK+01], and DBGlobe [PAP+03]. All three support relational data and the key set of relational database operations including joins and aggregation queries.

2.3 Distributed IR

We identify the following key characteristics and desirable performance features, which can greatly influence the key design choices for a P2P search engine.

1. *Peer Autonomy*: Peers work independently, possibly performing web crawls. There are two specific aspects of autonomy. First, whether a peer is willing to delegate the storage and maintenance of its index lists, agreeing that they be stored at other peers. For instance, a peer may insist on storing/maintaining its own index lists, worrying about possible problems, (e.g., index-list data integrity, security/privacy, availability, etc). Second, a peer may not be willing to dedicate substantial resources to other peers, e.g., store index lists produced by other peers.

2. *Sharing Granularity*: Influenced by the autonomy levels and performance concerns, the shared data can be at the level of complete index lists, portions of index lists, or even simply index list summaries appropriately to be defined.

3. *Ultra Scalability*: For the most popular terms, there may be a very large number of peers storing index lists. Accessing all such peers may not be an option. Therefore, a system design with ultra scalability in mind must foresee the development of mechanisms that can select the best possible subset of relevant peers, such that the efficiency of operation and result

quality remain acceptable. Another concern is that peers storing popular index lists may form bottlenecks hurting scalability. Thus, a design for ultra scalability also involves a novel strategy for distributing index list information that facilitates a large number of peers pulling together their resources during query execution, forming in essence large-capacity, "virtual" peers.

4. *Latency*: Latency may conflict with scalability. For example when, for scalability reasons, query processing may have to visit a number of peers which collectively form a large-capacity "virtual" peer, query response time may be adversely affected.

5. *Exact vs Approximate Results*: Approximate results may be justified at large scales. Recently, research results on high-quality approximate top-k algorithms have started emerging.

Within the field of P2P Web search, the following work is related to our efforts in building a P2P Web search engine.

Galanx [WGD03] is a P2P search engine implemented using the Apache HTTP server and BerkeleyDB. The Web site servers are the peers of this architecture; pages are stored only where they originate from.

PlanetP [CAPMN03] is a pub/sub service for P2P communities, supporting content ranking search. PlanetP distinguishes local indexes and a global index to describe all peers and their shared information. The global index is replicated using a gossiping algorithm. This system, however, appears to be limited to a relatively small number of peers (e.g., a few thousand).

Odissea [SMwW+03] assumes a two-layered search engine architecture with a global index structure distributed over the nodes in the system. A single node holds the complete, Web-scale, index for a given text term (i.e., keyword or word stem). Query execution uses a distributed version of Fagin's threshold algorithm [Fag02]. The system appears to create scalability and performance bottlenecks at the single-node where index lists are stored. Further, the presented query execution method seems limited to queries with at most two keywords. The paper actually advocates using a limited number of nodes, in the spirit of a server farm.

The system outlined in [RV03] uses a fully distributed inverted text index, in which every participant is responsible for a specific subset of terms and manages the respective index structures. Particular emphasis is put on minimizing the bandwidth used during multi-keyword searches. [LC03] considers content-based retrieval in hybrid P2P networks where a peer can either be a simple node or a directory node. Directory nodes serve as super-peers, which may possibly limit the scalability and self-organization of the overall system. The peer selection for forwarding queries is based on the Kullback-Leibler divergence between peer-specific statistical models of term distributions.

Rumorama [MEH05] is an approach based on the replication of peer data summaries via rumor spreading and multicast in a structured overlay. Ru-

morama achieves a hierarchization of PlanetP-like summary-based P2P-IR networks. In a Rumorama network, each peer views the network as a small PlanetP network with connections to peers that see other small PlanetP networks. Each peer can choose the size of the PlanetP network it wants to see according to its local processing power and bandwidth.

Alvis [LKP+06] is a prototype for scalable full-text P2P-IR using the notion of *highly discriminative keys* for indexing, which claims to overcome the scalability problem of single-term retrieval in structured P2P networks. Alvis is a fully-functional retrieval engine built on top of P-Grid. It provides distributed indexing, retrieval, and a content-based ranking module. While the index size is even larger than the single term index, the authors bring forward that storage is available in P2P systems as opposed to network bandwidth.

2.3.1 P2P Web Search with Minerva

We have developed a P2P Web search engine coined Minerva where we envision a network of peers that are crawling the web independently. Each peer (cf. Figure 2.3) maintains a local collection with a query processing engine which the peer can use to run queries locally. If the result quality is not satisfactory peers can use the information provided by remote peers. Minerva maintains a metadata directory that is layered on top of a DHT. It holds very compact, aggregated meta-information about the peers' local indexes and only to the extent that the individual peers are willing to disclose. A query initiator selects a few most promising peers based on their published per-term metadata. Subsequently, it forwards the complete query to the selected peers which execute the query locally. This query execution does not involve a distributed top-k query execution since each peer maintains a full-fledged local index with all information necessary to execute the query locally.

The computational model that we consider in this thesis is different from Minerva's architectural model: Instead of considering full-fledged search engines maintained by the peers we consider a network where single index-lists are spread across different peers. However, one could, in principle, create a distributed search engine by partitioning the index-lists across peers and running a distributed top-k algorithm to determine the top documents for a particular multi-term query, but this would cause major load imbalances since peers which maintain index-lists for popular terms will have to handle a lot of incoming request. In Chapter 8 we present the design of a search engine that organizes per-term index-lists in a way that avoids these bottlenecks. The data placement is determined by a hash function.

Minerva, however, makes a strong case for peer autonomy.

The novel aspects of the Minerva architecture are:

1. the way we leverage DHT-based overlay networks to build a directory service for efficiently managing and delivering novel metadata, consisting of compact, aggregated information that peers publish about their local indexes and

2. the way we use these metadata to appropriately select promising peers in order to limit the number of peers involved in a query (thus attaining high performance and improving scalability) while also ensuring high quality results.

Minerva was originally layered on top of a home-brewed re-implementation of Chord [SMK+01], which worked fine in the controlled settings of our lab experiments. In real-world deployments of Minerva, however, we often ran into system issues, e.g., caused by firewalls or strange IP configurations. Instead of reinventing the wheel, Minerva now uses Pastry [RD01] as the underlying routing mechanism and Past [DR01] for the persistent storage. A Peer in Minerva is implemented as a Pastry Application maintaining a *PastryNode*, i.e. Minerva implements the PastryApplication interface and is registered at a so called Pastry *Endpoint*. Once registered, the underlying PastryNode delivers incoming messages to the registered applications.

Figure 2.4 illustrates the Minerva approach. First, every peer publishes per-term summaries (*Posts*) of its local index to the directory. The DHT determines the peer currently responsible for this term. This peer maintains a *PeerList* of all postings for this term from across the network. Posts contain contact information about the peer who posted this summary together with statistics to calculate IR-style measures for a term (e.g., the size of the inverted list for the term, the maximum average score among the term's inverted list entries, or some other statistical measure). These statistics are used to support the query process, i.e., determining the most promising peers for a query.

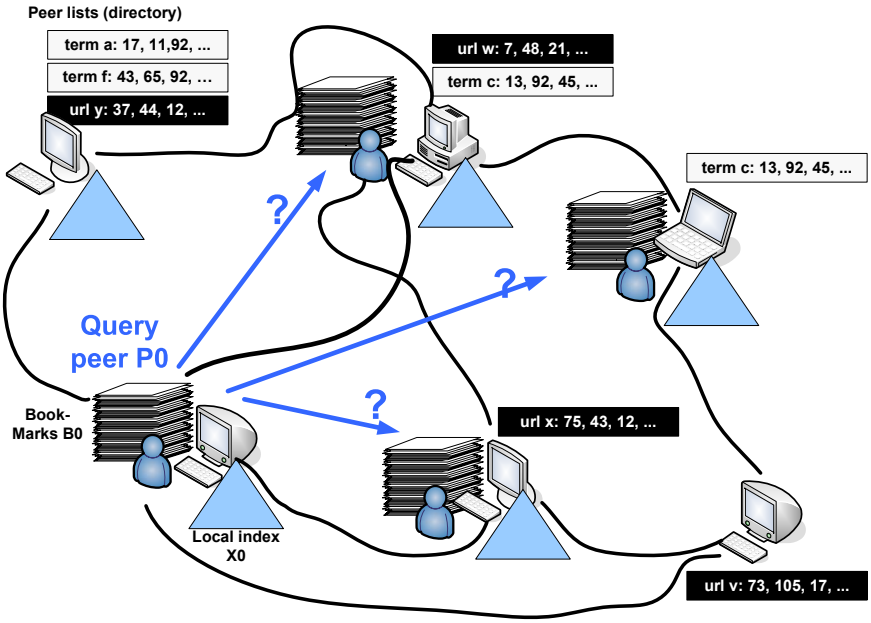

Figure 2.3: Minerva System Architecture

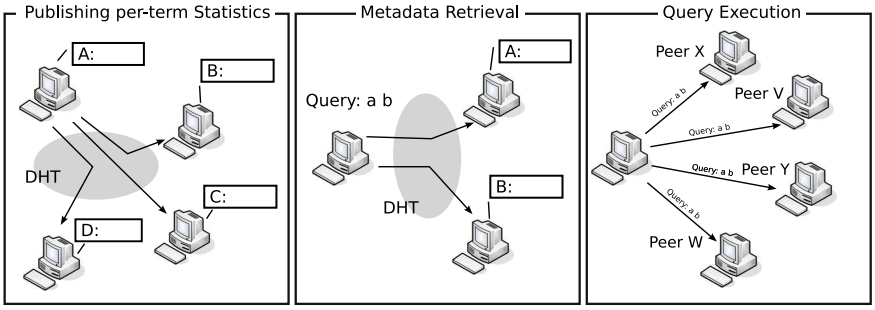

Figure 2.4: Metadata publication, retrieval, and query execution in Minerva

The querying process for a multi-term query proceeds as follows: first, the query is executed locally using the peer's local index. If the result is considered unsatisfactory by the user, the querying peer retrieves a list of potentially useful peers by issuing a *PeerList request* for each query term to the underlying overlay network, e.g. by executing a distributed top-k algorithm like [CW04, MTW05a]. A number of promising peers for the complete query is locally computed from these PeerLists. This step is referred to as *query routing*. Subsequently, the query is forwarded to these peers and executed based on their local indexes using a cutting-edge probabilistic top-k algorithm ([TWS04]). Note that this communication is done in a pairwise point-to-point manner between the peers, allowing for efficient communication and limiting the load on the DHT-based directory. Finally, the results from the various peers are combined at the querying peer into a single result list; this step is referred to as *result merging*.

2.3.2 Query Routing

Query Routing (also known as database selection) has been a research topic for many years, e.g. in distributed IR and metasearch [Cal00]. Typically, the expected result quality of a collection is estimated using precomputed statistics, and the collections are ranked accordingly. Most of these approaches, however, are not directly applicable in a true P2P environment. Within Minerva, we have adopted a number of popular existing approaches (which select peers based on how much they can improve the quality of collected results) to fit the requirements of our P2P environment and conducted extensive experiments in order to evaluate their performance [Ben07]. In addition, we have developed strategies which employ estimators of mutual overlap among the index lists of the peers selected to execute the query. Our result quality evaluation has shown that this approach can outperform other competing popular approaches based on quality estimation only, such as CORI [Cal00]. Taking overlap into account when performing query routing can drastically decrease the number of peers that have to be contacted in order to reach a satisfactory level of recall, which is a great step towards the feasibility of distributed P2P search.

2.3.3 Minerva at Document-Granularity

The DHT based directory is not limited to per-peer descriptions but can simply be turned into a full-document-index, i.e. the peers maintain the complete index lists at document-granularity. The mapping from terms to the responsible peers that maintain those index lists is given by the DHT as in the peer-granularity case.

In this scenario, we assume that peers are gathering information by e.g. crawling the web. These information are then indexed locally. Subsequently, peers are publishing (term, itemId, score)-triplets to the global index (cf. Figure 2.5). The terms are used as the keys for the DHT lookup. The peer which receives the (term, itemId, score)-triplets for a particular term creates an index list that is sorted by scores in descending order. For the actual query execution, the query initiator uses the DHT to find peers responsible for maintaining the index lists for the query terms. Then, the query initiator executes a distributed top-k query algorithm over these index lists.

However, as mentioned above, a system that follows this design would suffer from storage- and access-load imbalances. We will address this in Chapter 8.

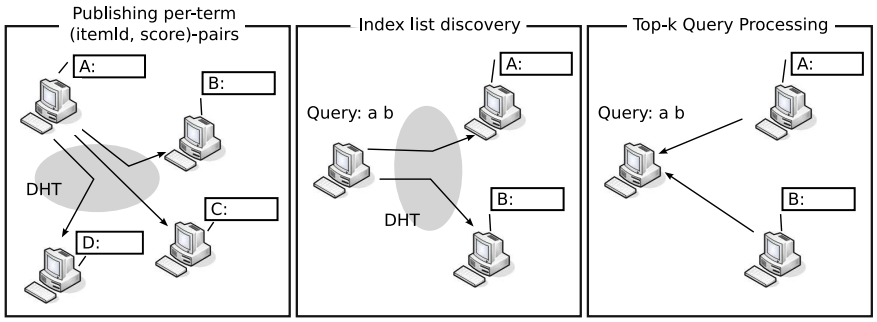

Figure 2.5: Minerva at document granularity

Chapter 3

State of the Art in Top-k Aggregation Query Processing

3.1 Introduction

Top-k query processing has received much attention in a variety of settings such as similarity search on multimedia data [CGM04, CGM04, Fag99, GBK00, BGRS99, NCS+01, dVMNK02], ranked retrieval on text and semi-structured documents in digital libraries and on the Web [AdKM01, LS03, TWS04, KKNR04, BJRS03, SCC+01, PZSD96, YSMQ01], spatial data analysis [BBK01, CP02, HS03], network and stream monitoring [BO03, KOT04, CW04] collaborative recommendation and preference queries on e-commerce product catalogs [YPM03, MBG04, BGM02, GBK01, CwH02], and ranking of SQL-style query results on structured data sources in general [ACDG03, CDHW04, BCG02]. [BCG02] addresses the mapping of top-k queries into range queries that can be handles by the query optimizer in a conventional RDBMS.

In terms of efficiency, the most successful approaches are based on the family of threshold algorithms (TA) originally developed by [FLN03, GBK00, NR99]. These techniques are fairly well understood for centralized data management, but much less explored for distributed systems such as peer-to-peer (P2P) federations [HCH+05] or sensor networks. For example, building a P2P Web search engine where thousands of nodes collaborate to provide Google functionality in a decentralized and self-organizing manner would be a great application for distributed top-k query processing.

3.2 Family of Threshold Algorithms

Among the ample work on top-k query processing (see the references in Section 3.1), the TA family of algorithms for monotonic score aggregation [FLN03,

GBK00, NR99] stands out as an extremely efficient and highly versatile method. In the following we shortly discuss three popular threshold algorithms.

The Threshold Algorithm (TA)

1. Do sorted access in parallel into each of the index lists L_i. If an item x has been seen in one of the lists, do a random access to retrieve the missing scores from all the other index lists. The scores are then aggregated using a monotone aggregation function (usually sum). Let $s(x)$ denote the final score. If $s(x)$ is currently among the k items with the highest score, remember x, otherwise drop it.

2. For each list L_i the algorithm remembers the score τ_i of the last item retrieved by sorted access. The aggregation τ of these scores τ_i defines the score upper-bound for the items that have not been observed to far.

3. The stopping condition is defined as follows. As soon as there have been seen at least k items with an aggregated score greater or equal than τ, then halt.

It is important to note that this algorithms completely evaluates the score of an item using random accesses as soon as this item has been observed.

NRA

NRA (aka. TA-sorted) variants process the (docID, score) entries of the relevant index lists in descending order of score values, using a simple round-robin scheduling strategy and making only sequential accesses on the index lists. TA-sorted maintains a priority queue of candidates and a current set of top-k results, both in memory. The algorithm maintains with each candidate or current top-k document d a score interval, with a lower bound worstscore(d) and an upper bound bestscore(d) for the true global score of d. The worstscore is the sum of all local scores that have been observed for d during the index scans. The bestscore is the sum of the worstscore and the last score values seen in all those lists where d has not yet been encountered. We denote the latter values by $high(i)$ for the ith index list; they are upper bounds for the best possible score in the still unvisited tails of the index lists. The current top-k are those documents with the k highest worstscores. A candidate d for which $bestscore(d) < min\text{-}k$ can be safely dismissed, where $min\text{-}k$ denotes the worstscore of the rank-k document in the current top-k. The algorithm terminates when the candidate queue is empty (and a virtual document that has not yet been seen in any index list and has a $bestscore \leq \sum_{i=1\ldots m} high(i)$ can not qualify for the top-k either).

Probabilistic Pruning

For approximating a top-k result with low error probability [TWS04], the conservative bestscores, with $high(i)$ values assumed for unknown scores, can be substituted by quantiles of the score distribution in the unvisited tails of the

index lists. Technically, this amounts to estimating the convolution of the unknown scores of a candidate. A candidate d can be dismissed if the probability that its bestscore can still exceed the *min-k* value drops below some threshold:

$$P[worstscore(d) + \sum_i S(i) > min\text{-}k] < \varepsilon$$

where the $S(i)$ are random variables for unknown scores and the sum ranges over all i in which d has not yet been encountered.

3.3 Top-k Query Processing by Generated Range Queries

[BCG02] addresses the mapping of top-k queries into range queries that can be handled by the query optimizer in a conventional RDBMS. A top-k query expressed in a SQL-like language could, for instance, look like this [CG96]:

```
SELECT * FROM R
      WHERE A1 = q1 AND ..... AND An = qn
      ORDER k BY Dist
```

A_1, A_2, ..., A_n are the attributes of relation R. The `ORDER BY` clause uses some distance function `Dist` to rank the tuples w.r.t. the given values q_1, q_2, ..., q_n. The parameter k determines the maximum size of the result ranking. More formally, given a query $q = (q_1, q_2, ..., q_n)$ and a tuple $t = (t_1, t_2, ..., t_n)$, it is assumed that the distance function $Dist(q, t)$ returns a positive real value. The paper considers only top-k queries over continuous-valued real attributes, and to distance functions that are based on vector p-norms, for instance,

$$Sum(q,t) = \|q - t\|_1 = \sum_{i=1}^{n} |q_i - t_i|$$
$$Max(q,t) = \|q - t\|_\infty = \max_{i=1}^{n} |q_i - t_i|$$

The mapping algorithm is not designed to be a stand-alone top-k algorithm. It can be seen as a plugin for existing RDBMS to be able to efficiently handle top-k queries using a transformation of top-k queries into range queries. Bruno et al. propose the usage of multi-dimensional histograms to detect a region in the data-space that contains the best tuples for the given distance function. More precisely, the query processing consists of the following three steps:

1. For a given query q, use a multidimensional histogram so detect a distance d_q such that the region around q, that contains all tuples t with $Dist(q, t) \leq d_q$, is expected to contain k tuples.

2. Retrieve the tuples in the previously determined region using a range query.

3. If there are less than k tuples included in the region, increase the distance d_q and re-start the query. Otherwise, rank the retrieved tuples according to the distance function and return the top-k results.

In our work we use aggregation functions rather than distance functions. The two concepts, however, are equivalent if the attribute-values are normalized to a particular range, e.g. to $[0, 1]$.

Recently, Cao and Wang [CW04] used the idea of transforming a top-k query into a range query in their TPUT algorithm that efficiently processes top-k queries in distributed systems.

3.4 Top-k Queries over Distributed Data Sources

The first distributed TA-style algorithm has been presented in [BGM02, MBG04]. The emphasis of that work was on top-k queries over Internet data sources for recommendation services (e.g., restaurant ratings, street finders). Because of functional limitations and specific costs of data sources, the approach used a hybrid algorithm that allowed both sorted and random access but tried to avoid random accesses. Scheduling strategies for random accesses to resolve expensive predicates were addressed also in [CwH02]. In our widely distributed setting, none of these scheduling methods are relevant as they still incur an unbounded number of message rounds. The method in [SMwW+03] addresses P2P-style distributed top-k queries but considers only the case of two index lists distributed over two peers. Its key idea is to allow the two cohort peers to directly exchange score and candidate information rather than communicating only via the query initiator. Unfortunately, it is unclear and left as an open issue how to generalize to more than two peers.

In contrast, state-of-the-art algorithms for distributed top-k aggregation use a fixed number of communication rounds to bound latency and aim to minimize the total network bandwidth consumption. The first algorithm in this family was the *TPUT* (Three-Phase Uniform Threshold) algorithm [CW04], in which a query coordinator, typically the network node which initiates the query, executes a three-phase distributed threshold algorithm. Section 3.5 presents TPUT in more detail. *TPAT* [YLW+05] is a modification of TPUT where the threshold, that is the same for all index lists, is adapted to the specifics of the value distributions; however, the authors state that their solution may incur infeasible computational cost. In Chapter 6 we will also consider the issue of adaptive thresholds and introduce an efficient way to calculate them.

A special topology is considered in [BNST05], where the authors address the optimization of communication costs for top-k queries in a P2P network with a hypercube topology, focusing on efficient routing and caching in a network with dedicated super peers. A three-phase threshold algorithm for distributed sensor networks with a hierarchical topology similar to TPUT is presented in

[ZYVG+05], but exploits the given hierarchy to compute a better lower bound in the first phase. Unlike these approaches, the algorithms that we present in this work, as well as TPUT have been designed for general networks without assumptions on specific topologies.

Queries that find all items whose aggregated value is greater than a specified threshold (which can be seen as a "dual problem" to top-k querying) are addressed in [ZOWX06] using sampling to reduce the communication overhead. The algorithm samples data items in *each* node and sends them to the coordinating node, which has a suboptimal effect on bandwidth consumption.

The work presented in [MSDO05] considers the related problem of finding frequent items in distributed data streams within fixed time intervals, exploiting the hierarchical structure of the communication network. The more general problem of continuous top-k or threshold queries in an environment of distributed streams, like distributed monitoring of aggregated values [BO03, KCR06, SSK06], is outside the scope of this thesis.

[APV06] introduces a top-k algorithm for unstructured Peer-to-Peer systems, where the query is broadcasted into the network and executed locally at each peer. They propose a pruning technique on the route back to the query initiator where each intermediate node merges results from its child nodes and forwards only the best k items to its parent. This is not applicable in a setting where each peer delivers only partial scores and the final score is computed by summation, for example in document retrieval, as it cannot be guaranteed that the global top-k items after aggregation are encountered. This is due to the intermediate pruning steps. In Chapter 6, we address this problem by propagating individual score thresholds to each node to guarantee an exact and efficient computation of the global top-k query. Additionally, we optimize the tree structure beyond the random structure created by the flooding process.

3.5 Three Phase Uniform Threshold Algorithm (TPUT)

Cao and Wang [CW04] proposed an algorithm that efficiently calculates the exact top-k result in three phases. The main idea is to transform the top-k query into a range query where the range is determined via an estimation of the min-k value.

1. min-k **estimation phase:** (cf. lines $3-8$ in Algorithm 3.1) The query initiator P_{init} retrieves the top k items from each of the input index-lists. Subsequently, P_{init} calculates the worstscore for all observed items and ranks them accordingly. The worstscore of the item currently at rank k is min-k.

2. **Candidate retrieval phase:** (cf. lines $9-20$ Algorithm 3.1) Based on the min-k estimation, TPUT sends the min-k/m threshold to all involved peers that send back all $(itemId, score)$-pairs with $score \geq min\text{-}k/m$.

This ensures that all candidates (potential members of the final top-k result) have been found, in at least one of the lists. After the *min-k* value has been re-calculated, TPUT throws away all items with *bestscore* < *min-k*.

3. **Missing scores lookup phase:** (cf. lines $21-26$ Algorithm 3.1) For all the remaining candidates, TPUT looks up the missing scores by sending to each peer P_i a list of the candidates that have not been seen in list L_i so far. P_{init} can now calculate the exact score for all candidates, i.e. the true top-k results have been identified.

Phase 1			**Phase 2**			**Phase 3**		
$(a, 12)$	$(b, 8)$	$(a, 17)$	$(a, 12)$	$(b, 8)$	$(a, 17)$	$(a, 12)$	$(b, 8)$	$(a, 17)$
$(b, 10)$	$(c, 7)$	$(z, 13)$	$(b, 10)$	$(c, 7)$	$(z, 13)$	$(b, 10)$	$(c, 7)$	$(z, 13)$
$(c, 8)$	$(e, 6)$	$(e, 11)$	$(c, 8)$	$(e, 6)$	$(e, 11)$	$(c, 8)$	$(e, 6)$	$(e, 11)$
$(d, 6)$	$(z, 4)$	$(f, 10)$	$(d, 6)$	$(z, 4)$	$(f, 10)$	$(d, 6)$	$(z, 4)$	$(f, 10)$
$(e, 3)$	$(m, 2)$	$(c, 6)$	$(e, 3)$	$(m, 2)$	$(c, 6)$	$(e, 3)$	$(m, 2)$	$(c, 6)$
$(h, 3)$	$(g, 2)$	$(r, 5)$	$(h, 3)$	$(g, 2)$	$(r, 5)$	$(h, 3)$	$(g, 2)$	$(r, 5)$
$(f, 2)$	$(o, 1)$	$(b, 5)$	$(f, 2)$	$(o, 1)$	$(b, 5)$	$(f, 2)$	$(o, 1)$	$(b, 5)$

Table 3.1: Sample TPUT execution for a top-2 query: Phase 1 (left): Retrieve the top-2 items from each list. Phase 2 (middle): Retrieve all items with score above *min-k*$/3 = 6$. Phase 3 (right): Retrieve missing score via random lookups.

Table 3.1 shows an example of a top-2 query execution over 3 index lists. In Phase 1, the query initiator P_{init} retrieves the top-2 entries from each list and calculates the worstscore for all discovered items. This results in a ranking $((a, 29), (b, 18), (z, 13), (c, 7))$. The item at rank 2 is currently b with a score of 18. At this stage we cannot throw away the items z and c as both have a bestscore above *min-k*: bestscore(z) = worstscore(z) + 10 + 7 = 13+10+7 = 30, bestscore(c) = worstscore(c) + 10 + 13 = 7+10+13 = 30. In Phase 2, P_{init} retrieves from each list all $(itemId, score)$-pairs that have a score above or equal *min-k*$/m$, i.e. a score greater or equal to 6. With these new information, P_{init} re-calculates the worstscores for the known items. This results in a new ranking $((a, 29), (c, 21), (b, 18), (e, 17), (z, 13), (f, 10), (d, 6))$. Now, *min-k* $= 21$ and its known that the scores in the tails of the index lists are not bigger than 5. With this knowledge, P_{init} can prune away f ($worstscore(f) = 10$, $bestscore(f) = 20$) and d ($worstscore(d) = 6$, $bestscore(d) = 16$) because their bestscores are below *min-k*. Phase 3: P_{init} retrieved the missing scores for the currently not fully evaluated candidates $\{a, b, e, z\}$ (c is already fully evaluated). Subsequently, P_{init} re-calculates the ranking, thus identifies the final top-2 result-list $((a, 29), (b, 26))$. Figure 3.1 presents an illustration of TPUT's phase structure.

Algorithm 3.1 TPUT

1: **input:** list of peers to be queried L, k
2: **output:** $TopK$ list
3: **for** $i = 1$ to $L.length$ in parallel **do**
4: result[i] = L[i].getTopK()
5: **end for**
6: $L_{agg} = \bigcup_i^+ result[i]$
7: $L_{sorted} = \rho_{worstscoredesc}(L_{agg})$
8: $min\text{-}k = L_{sorted}[k]$
9: **for** $i = 1$ to $L.length$ in parallel **do**
10: result[i] = L[i].getDocumentsAbove($min\text{-}k/L.length$)
11: **end for**
12: $L_{agg} = L_{agg} \cup^+ \bigcup_i^+ result[i]$
13: $L_{sorted} = \rho_{worstscoredesc}(L_{agg})$
14: $min\text{-}k = L_{sorted}[k]$
15: $C = \emptyset$
16: **for** $i = 1$ to $L_{agg}.length$ **do**
17: **if** $L_{agg}[i].bestscore \geq min\text{-}k$ **then**
18: C.add($L_{agg}[i]$)
19: **end if**
20: **end for**
21: **for** $i = 1$ to $L.length$ in parallel **do**
22: result[i] = L[i].getMissingScores(C)
23: **end for**
24: $L_{agg} = C \cup^+ \bigcup_i^+ result[i]$
25: $L_{sorted} = \rho_{worstscoredesc}(L_{agg})$
26: $TopK = L_{sorted}.sublist(k)$
27: **return** TopK

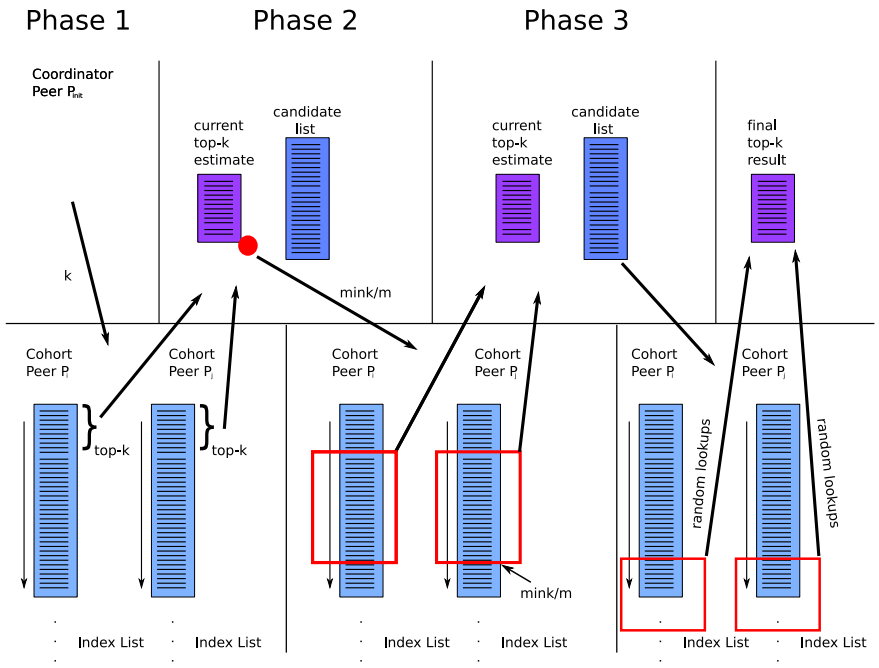

Figure 3.1: TPUT

3.6 Exact vs. Approximate Algorithms

Some of the above algorithms are *exact* in the sense that they compute the exact top-k results, whereas others are *approximate* as they may deviate from the exact top-k results with low probability. For example, TPUT is exact, while our own algorithm KLEE that we present in the following Chapter is approximate. Omitting its third phase makes TPUT become approximate, too. In this case, it would output the top-k result candidates at the end of phase 2 as an approximate result (i.e., assuming all unknown values for these items are zero).

On the other hand, we can turn all approximate algorithms of this kind into exact ones by adding an additional random-access phase to resolve the uncertainty between lower and upper bounds of the aggregated values for the result candidates. In contrast to the third phase of TPUT method, this phase does not necessarily have to look up all unknown values, but can apply the following more efficient greedy heuristics. Consider candidates x in descending order of their current lower bounds $lb(x)$ for their aggregated values. Generate a random access for x on the list L_j with the highest possible increment of the aggregated value for x, based on the last value retrieved from the list. When a certain number of random lookups have been generated, group these into batches, so that all requests to the same network node can be combined into one message. Perform these batched lookups, then re-compute lower and upper

bounds of aggregated values, re-compute the *min-k* threshold, and eliminate candidates. The procedure is repeated until only k candidates survive.

Chapter 4

The KLEE Algorithm

This chapter is based on our own work in [MTW05a] and presents a novel family of algorithms for distributed top-k query processing, coined KLEE. The name of the algorithm refers to the German name of the plant known as clover in English. Clover usually has three leaves but infrequently occurs with four leaves. Our KLEE algorithm uses three or, optionally for additional optimization, four algorithmic steps[1]. The most relevant prior work [CW04] provided a distributed top-k algorithm with a small, fixed number of (only three) communication phases, ensuring small query response times. We also adopt the requirement for a small number of communication phases. However, KLEE goes far beyond. The salient features and novel contributions of KLEE are the following:

- KLEE comes with two flavors, one involving only two and one involving three communication phases. It recognizes that the number of communication phases is only one aspect of guaranteeing short response times, which, in turn, is only one aspect of overall efficiency. In particular, as limited network and IO bandwidth appear to be key contributors to response times, KLEE ensures that significantly smaller messages are exchanged and that random IOs at participating peers are avoided, resulting in strong gains in response time and network bandwidth and lighter peer loads compared to TPUT.
- KLEE is the first to make a strong case for approximate top-k algorithms for wide-area networks, showing how significant performance benefits can be enjoyed, at only small penalties in result quality.
- KLEE provides a flexible framework for top-k algorithms, allowing for trading-off efficiency versus result quality and bandwidth savings versus the number of communication phases.
- We have implemented KLEE and a number of competing algorithms and conducted comprehensive experimental performance evaluation using real-

[1] The name Klee also refers to Paul Klee who was a Swiss painter of German nationality

world and synthetic data, which shows the consistent superiority of KLEE over its competitors.

- KLEE is equipped with various fine-tuning parameters and we provide a discussion of how these can be automatically adjusted to underlying data and system characteristics.

4.1 Key Ideas and Data Structures

The proposed approach is based on having a per-query coordinator peer and a set of cohort peers. In our setting, the coordinating peer is the peer where the query was initiated, P_{init}. The cohort peers, are the peers storing the index lists, based on which the document scores will be computed. The algorithm is structured to proceed in a number of phases, with each phase consisting of a round-trip communication between the coordinator and the cohorts. In general, in each phase, the coordinator requests and receives from each peer a portion of the peer's local index information, which permits the coordinator to run a top-k algorithm (such as the TA algorithm or variants) based on the collected information about the peers' index lists.

4.1.1 The HistogramBlooms Structure

In KLEE, each peer maintains a set of statistical metadata describing its index list. In particular, histogram-based information is maintained to describe the distribution of scores in the index list. The range of possible score values cover the range $(0, 1]$. For simplicity, we assume that peer histograms are equi-width, consisting of n cells, each cell being responsible for $(1/n)th$ of the score range. It would be straightforward to employ other forms of histograms. Associated with each cell i, each peer maintains the following information

- The lower and upper values, $lb[i]$, $ub[i]$, respectively, defining the range of scores being covered by this cell,
- The value of $freq[i]$, defining the number of document IDs whose scores in the peer's index list fall within $lb[i]$ and $ub[i]$,
- The average score, $avg[i]$, computed over all scores in the cell, and
- A synopsis of the document IDs whose scores fall in this cell, $filter[i]$. In particular, this compact representation is constructed using Bloom filters.

Bloom filters have received a lot of attention in our community, given their distinguishing ability to, on the one hand, represent compactly the contents of a set and, on the other, efficiently test whether a given item is a member of the set. Briefly, in their simplest form, Bloom filters work as follows: a bitmap V containing b bits, initially all set to 0, is used to compact the information in a set $S = \{a_1, a_2, ..., a_s\}$. Each value of set S is hashed into V. In general h independent hash functions, $h_1, h_2, ..., h_h$ can be used for each element of S

producing h values, each varying from 1 to b and setting the corresponding bit in vector V. Testing if an element e belongs to set S is now very fast: simply, the same h hash functions are applied on e and the bits of V in positions of $h_1(e), h_2(e), ..., h_h(e)$ are checked. If at least one of these bits is 0, then e does not belong to S. Else, it is conjectured that e belongs to S, although this may be wrong (this is referred to as a "false positive"). Given the number of items, s, of the set for which a filter is created, which set a number of bits in the filter, by tuning h and b one can control the probability for false positives, which is given by

$$PFP \approx (1 - e^{-hs/b})^h$$

[Blo70, FCAB98], where s is the number of values in the set S, b is the size of the filter/bitmap, and h is the number of hash functions. When $h = 1$, the term b/s, coined the load factor, controls PFP.

Figure 4.1 shows a sample usage of a Bloom filter: Two items, x_1 and x_2 are inserted into the initially empty Bloom filter. Subsequently, when issuing a membership test for y_1 and y_2 we see that y_1 is not contained in the filter since not all corresponding bits are set, whereas y_2 seems to be contained in the filter, i.e. it is contained with high probability or it is a false positive. [BM05] gives an overview of the usage of Bloom filters.

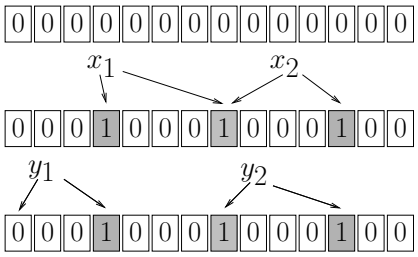

Figure 4.1: Insertion of two elements in a Bloom filter. y_1 is not contained in the filter. y_2 is either in the filter or it is a false positive.

As mentioned, KLEE uses Bloom filters to compactly represent, for each histogram cell, the set of documents whose scores fall in this cell. This information, coupled with the statistical metadata, can prove of great value to the coordinator to compute a high quality top-k approximation swiftly and efficiently.

4.1.2 Harvesting HistogramBlooms

In the first phase, at the coordinator's request, each cohort peer replies with its local top-k list, and a fraction of its HistogramBlooms data structure. The coordinator then can address the missing-scores problem as follows: for every peer P_i that has not reported a score for docID, using the Bloom-filter cell summaries of P_i and the hash functions, it can find to which histogram cell of peer P_i the docID belongs say c, (by simply testing for membership of docID in the filters of each cell, and stopping when a test is successful). Then, it can use the

average score associated with that histogram cell, $avg[c]$, to replace the missing score of P_i for docID. The missing-documents problem can then be dealt with as follows: The coordinator, having attacked the missing-scores problem, can then produce an approximation of the top-k result and identify the k-th total score in this top-k approximation, *min-k*. Thus, a per-peer candidate list can be constructed, consisting of all the docIDs (and their scores) that locally in a peer have a score that is greater than *min-k*$/m$. Each of the m cohort peers then can be asked to send its candidate list. After receiving this information, the coordinator can then compute a higher-quality top-k approximation. Intuitively, the HistogramBlooms structure allows the coordinator of the algorithm the chance to gather score information from deep enough into the index lists of the cohort peers, without paying the bandwidth cost of retrieving long subsets of the peers' index lists.

4.1.3 The Candidate Filters Matrix (CFM)

The above solution to the missing-documents problem, although helpful, may require further optimization. At the end of the 1st phase, the coordinator has qualitative information at its disposal that allows it to estimate how good its top-k score approximation is. For instance, if too many missing values are replaced by averages from "low-end" ("high-end") peer-histogram cells, then the approximation is with high probability of low (high) quality. In addition, and perhaps more importantly, even if the *min-k* approximation at the end of the first phase is accurate, it is possible that the per-peer candidate lists sent by the peers in the second phase will be much longer than needed, wasting thus a lot of bandwidth. The reason is that, the value *min-k*$/m$, especially for larger values of m, may be very small, and a very large fraction of the docIDs at each peer may have a higher score. For these reasons, an additional "candidate list reduction" phase may be employed to avoid high network bandwidth overheads. The central insight is to gather information about the contents of the per-peer candidate lists so that only docIDs that belong to "enough" candidate lists (and have a chance to have a TotalScore higher than *min-k*) are sent; the rest will be filtered out and not sent. In this phase, the peers will:

1. each identify the contents of its candidate list set, i.e., find those docIDs associated locally with a score that is better than (*min-k*$/m$) and

2. create a bitmap filter of this set, called the peer's Candidate Filter, CF. Specifically, for each docID with $score(docID) >$ (*min-k*$/m$), the peer will hash the docID and set the proper bit in its CF.

Utilizing the histogram statistics received, P_{init} can roughly know from the 1st phase the number of documents at each peer that have a better score than *min-k*$/m$. The maximum of these numbers will be sent to the peers and will be used by them in the bitmap construction so that all peers' CFs will have the same size, b. When P_{init} receives these CFs it constructs a bitmap matrix, the CF Matrix (CFM). The CF Matrix:

- is an $m \times b$ matrix,
- its i-th row is the CF received from the i-th peer.

4.1.4 Harvesting Candidate List Filters

The rationale for building the *CF Matrix* is that, by construction, all docIDs (from all m peers) which have a higher score than the $min\text{-}k/m$ in R of the m peers, will be hashed into a column of the CF Matrix with R bit positions set. The central conclusion that can now be drawn is that the docIDs that hashed into columns with a small number of set bits, need not be sent, since they have a better score than $min\text{-}k/m$ in only a small number of peers, making the likelihood of these docIDs having a total score better than $min\text{-}k$ very small. Thus, for appropriately selected values of R (e.g. for a majority of the peers) the docIDs that hashed into columns of the CF Matrix which have R bits set, need be sent only. In this way, P_{init} can substantially reduce the size of the set of (docID, score) pairs which peers will be asked to send, yielding obvious bandwidth benefits. Associated with the construction and exploitation of the CF Matrix, there are three challenges:

1. obtain the needed information with low network bandwidth overhead, while
2. avoiding extensive filtering of docIDs that would reduce the quality of the top-k list result, and
3. being able to estimate the expected benefits of producing and exploiting Candidate Filters before hand, so to avoid having an additional communication phase if they are not needed.

4.2 The KLEE Algorithmic Framework

4.2.1 The Peer Cohorts' Preparation

Each peer, given its sorted index list, constructs the HistogramBlooms structure described previously. The construction of the histogram-related data is straightforward. The construction of the per-histogram-cell filters is also simple: In the same scan of the index list needed to construct the histogram data, for each histogram cell, a set, cell-docID-set, is created whose elements are the docIDs belonging to this cell. For each such i, $cell - docID - set[i]$ a Bloom filter, $filter[i]$, is constructed. All peers use the same number of and the same hash functions for the $filter[i]$ construction, for all i. However, different peers, in general, will be expected to have histogram cells of different sizes. Therefore, the size of the filters $filter[i]$ at different peers will of course be different, driven primarily of the need to ensure a low probability for false positives. Since the construction of the histograms and related filters may be time-consuming, these can be precomputed and stored locally at each peer, to avoid incurring the overhead of computing these "on line".

4.2.2 KLEE: A High-Level View

When a query $q(T, k)$ is initiated at a peer, P_{init}, this peer assumes the responsibility for coordinating the execution of the top-k algorithm, communicating with the m cohort peers with relevant index lists for the terms in T. The algorithm has in general the following four steps:

1. The Exploration Step (cf. Figure 4.2). P_{init} communicates with the m cohort peers in order to produce a good estimation of the *min-k*, which in turn yields the per-peer candidate lists. For a peer P_i its candidate list is defined to contain those docIDs for which score(docID) $>$ (*min-k*/m).

2. The Optimization step. This step is performed by P_{init} locally. It analytically estimates the expected benefits from engaging a Candidate List Reduction phase, by arguing about the expected values in the candidate filters that would be constructed by the cohort peers.

3. The Candidate Reduction Step. This step is optional, in the sense that it is executed only when indicated by the previous step. It requires one round-trip communication phase with the cohorts to construct the CFM data structure. Using the latter, a new set of per-peer candidates are constructed, replacing the ones constructed in the first step. Specifically, for a peer P_i its candidates are those docIDs for which hash(docID) is one of the columns of the CFM with enough bits set.

4. The Candidate Retrieval Step. This consists of a final round-trip communication phase with the cohorts to obtain their candidate lists and compute the final top-k result.

Note that the optimization step acts basically as a point for trading-off bandwidth performance vs the number of communication phases. This step predicts the potential bandwidth savings resulting from the candidate list reduction; these, in turn, can be weighed against the cost in latency of engaging an additional round-trip communication phase with the peers. Different decisions can be made, depending on which metric is considered to be more critical. In the following subsections each step of the framework is presented in detail.

4.2.3 The Exploration Step

This is the first step of KLEE embodying the first coordinator-cohorts communication phase. It addresses the missing-scores problem as follows:

1. P_{init} sends a "start" request with the query $q(T, k)$.

2. Peers respond with:

   a. their local top-k lists,

   b. for each of the c "high-end" cells (i.e. for the cells covering up to, say the top few percent of the highest scored documents): the histogram-cell information ($freq[i]$, $lb[i]$, $ub[i]$, $avg[i]$, and $filter[i]$), $i = 1, ..., c$.

c. for each of the remaining i, $i = c + 1, ..., n$, "low-end" cells: $freq[i]$, and $avg[i]$.

3. P_{init} then approximates the top-k list, as follows:

   a. When the score of some document with $docID_i$ is missing in some index list $I_j(t)$, P_{init} hashes $docID_i$ and checks for membership in the $filter[r]$, $r = 1, ..., c$ (i.e., in the per-cell document filters sent by peer P_j) to find out to which histogram cell in P_j $docID_i$ belongs. The check stops when either a membership test is successful, or until all available $filter[r]$ summaries are exhausted.

   b. If $docID_i$ is found to be a member of, say, $filter[r]$, P_{init} uses the average score associated with that cell, $avg[r]$, to replace the missing score.

   c. Else, P_{init} replaces the missing score with a weighted average score computed using the frequencies and average scores associated with the 'low-end' cells of P_j.

   d. This process is repeated for all docIDs for which scores are missing and for all P_j from which scores are missing.

4. Having replaced all missing scores, P_{init} computes the top-k list approximation and identifies the score of the k-th document in this list as the *min-k*.

5. Furthermore, given *min-k*, implicitly defines the candidate list of each peer as follows: The CandidateList of peer P_j is defined to be the set:

$$\{docId : docId \in I_j \wedge score(docId) > min\text{-}k/m\}$$

4.2.4 The Optimization Step

This is the second step of KLEE. It requires no communication; it is executed completely locally within P_{init}. The main task here is to analytically estimate the expected bandwidth savings resulting from possibly employing the candidate list reduction phase. Thus, we derive the fundamental relation that yields these expected savings and the parameters it depends on. The analysis uses the value d, defined as the average size of the peer candidate lists (that is, the average number over all peers of docIDs having a score that is greater than $min\text{-}k/m$, at the end of phase 1). For clarity, we assume that the probability of false positives is made very small, using appropriate load factors, so approximating the average number of (docID, score) pairs sent by each peer with d, is acceptable; actually, these probabilities are not hard to compute, but would make the presentation harder to follow. Recall that for the CF construction, peers use just one hash function. Arguing about the expected values of the CF Matrix, we note that the probability of any bit of a column being set (independently by a peer in its CF) is given by $P_1 = 1/lf$ where, lf is the load factor for the Bloom filter which

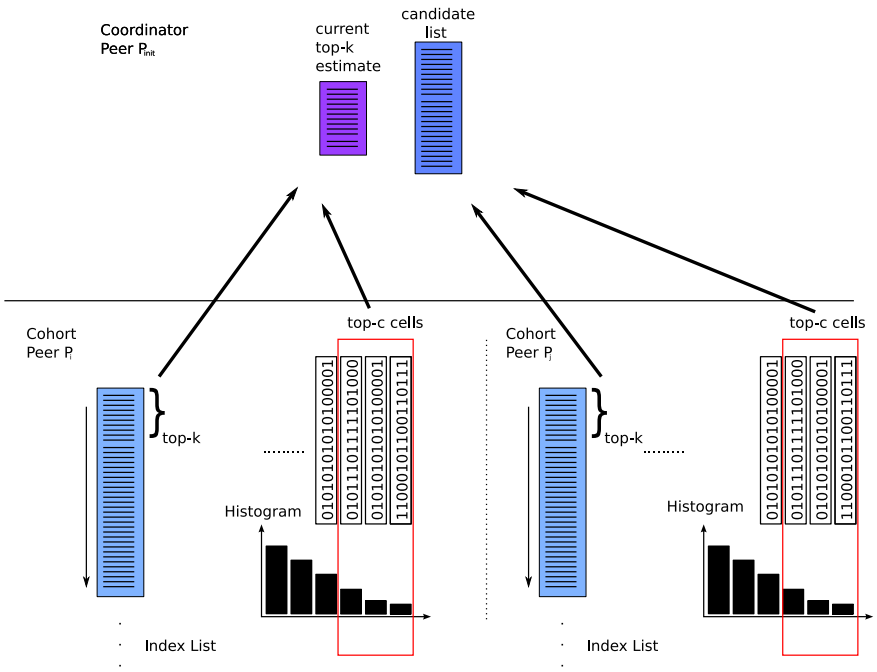

Figure 4.2: Two peers responding to P_{init}

is given by: $lf = b/d$ where b is the size of the peers' CFs. Next, the key value to estimate is the expected number of columns of the CF Matrix which have at least R bit positions set. The term P_R refers to the probability of any column satisfying this criterion. P_R is given by the following binomial distribution:

$$P_R = \sum_{i=R}^{m} \binom{m}{i} \times (\frac{1}{lf})^i \times (\frac{lf-1}{lf})^{m-i}$$

The bandwidth cost, measured in terms of the number of (docID, score) pairs sent by all peers, in the final phase of KLEE without the Candidate List reduction phase, C, is given by $C = d \times m$. The bandwidth cost in the version of KLEE with the candidate list reduction phase engaged, Cr, consists of the cost of sending the candidate list filters at phase 2, $C_{r,2}$ and the cost of sending the (docID, score) pairs in the final phase 3, $C_{r,3}$. For the latter cost, recall that P_{init} sends to the peers in the phase 3 the column indices which are found to satisfy the criterion that at least R bits are set and that each peer responds only with the docIDs that hash into these positions. Thus, we need to compute the probability that in each peer CF there is a bit set for the specific indices sent by P_{init}. $C_{r,3}$ is thus given by $C_{r,3} = P_R \times d \times m$ since in each peer's CF, a bit position belongs to a column with at least R bits set with probability P_R, and since there are d bits set in each peer, and there are m peers in total. Comparing C_o and $C_{r,3}$ we see that $C_{r,3} = PR \times C_0$ making the value of P_R the key to the expected savings in the bandwidth in the last phase of the algo-

rithm. The actual costs C_o and C_r must be multiplied by the average number of bytes required for each (docID, score) pair. Additionally, the cost of sending the candidate list filters, $C_{r,2}$, must also be accounted for. This cost is simply given by $C_{r,2} = (m \times b/8)$ bytes.

4.2.5 The Candidate Reduction Step

The following details step 3 of KLEE (cf. 4.3), which revolves around the construction and manipulation of the peers' CF structures. Candidate reduction: Improving the quality of the top-k approximation and addressing the missing documents problem:

1. P_{init} first refines the set candidate_list(P) for a peer, P, to be all docIDs that:
   - P has not sent to P_{init} so far and
   - have a score in the index list of P that is greater than the minimum score of the histogram cell holding the value $min\text{-}k/m$.
2. P_{init} computes the size of candidate_list(P_i) for each peer P_i, based on the histogram data received in step 1 and then finds their maximum, max_size_candidate_list. Then,
   - P_{init} sends to each peer P_i the current top-k estimate and max_size_candidate_list,
   - Each peer P_i, computes and returns to P_{init}:
   - The CF: using just one hash function and a bitmap with size $b = load\_factor \times max\_size\_candidate\_list$, with a load_factor value large enough to ensure low probabilities of false positives. The CF is constructed by hashing each docID of its candidate list into this bitmap, and
   - the true scores of the docIDs in the top-k estimate.
3. P_{init} constructs the CF bit matrix, CFM, of size $m \times b$. As mentioned, the rows in this matrix are the CF filters received from the peers: $CFM[i, j]$ represents the j-th entry in peer P_i's CF filter for candidate_docs(P_i).
4. P_{init} defines the interesting columns of its CFM to be the indices of those columns with at least a number R of bits set.
5. Finally, P_{init} redefines the candidate list of a peer P_i to be the subset of P_i's original candidate list consisting of only the docIDs that hash into the interesting columns of P_i's CF.

As mentioned, by construction, after phase 2, all docIDs which have a higher score than the ($min\text{-}k/m$) in R peers, will be hashed into a column of CFM with R entries set. The converse, however, does not necessarily hold; i.e. when two different bit positions in a column of CFM are set, they may either come from

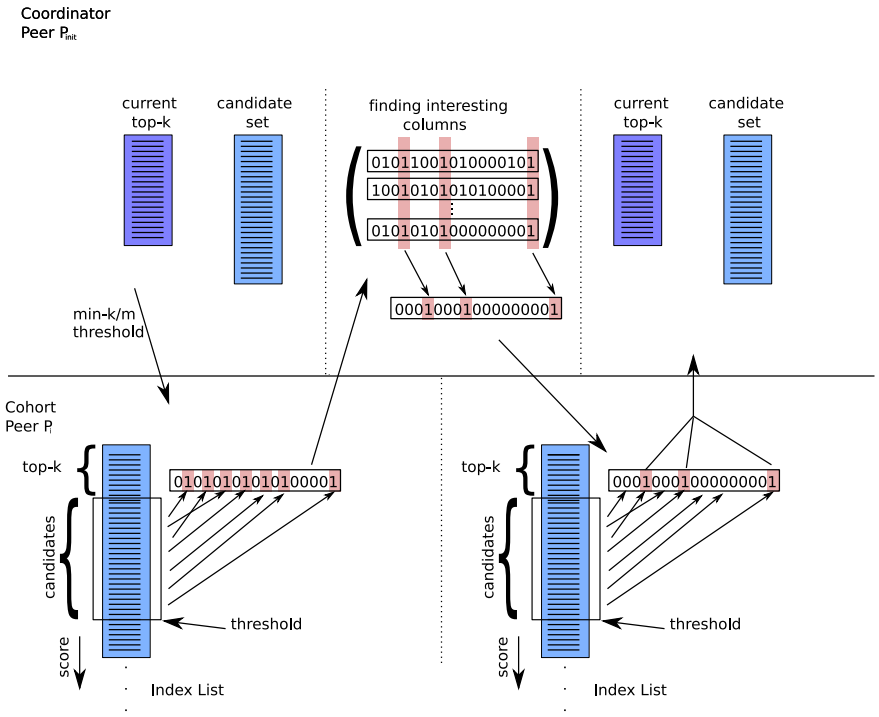

Figure 4.3: Constructing CFM from CFs

the same docID known to the respective peers, or from two different docIDs that happened to hash into the same bit position. This obviously implies that these false positives introduced by the CF filters of the different peers will lead to having peers send more docIDs than absolutely necessary in the next phase. This problem is in essence the false positives problem and can be addressed by appropriate settings of the values of the load factor for the filter construction.

4.2.6 The Candidate Retrieval Step

This is the final step and represents the final communication phase between the coordinator and the cohorts.

1. P_{init} asks and receives from each peer P_j the $(docID, score)$ pairs, for each $docID$ that belongs in P_j's candidate list, as the latter is defined either from step 1 or from step 3.

2. P_{init} then calculates the new top-k list result, based on the $(docID, score)$ pairs received. In essence, with the 4-step version of the algorithm, peers are asked to perform some more processing, introducing a trade-off between top-k approximation latency and peer resource utilization, on the one hand, and overall network bandwidth on the other.

4.3 KLEE Parameters

The main parameters characterizing the functionality offered by KLEE are:

(*i*) the number of cells, c, for which filters are sent by each peer in the first step

(*ii*) the number of bits, R, that have to be set in order for any column of the CFM to be considered as interesting by the coordinator in the third step.

KLEE also utilizes parameters pertaining to the construction of the histogram-cell Bloom filters and in the construction of the CFs at peers; these parameters are the load factor and the number of hash functions to be used so that, given the number of entries, the probability of false positives is kept below an acceptable threshold value. The values for the latter parameters, however, are well understood from the related literature and do not deserve further attention.

A good choice of the parameter c depends on the skew of the score distributions. We employ a technique that bounds the score-prediction error that we make by fetching only the top c histogram cells compared to the entire histogram.

Defining the right value for the parameter R, which represents the number of bits that need be set in order for a column of CFM to be considered interesting in step 3, may be error-prone. A key insight would be to utilize the histogram data available at P_{init}. Instead of simply counting set bits in the columns of CFM, we could multiply each set bit with an appropriately-selected score value from the peers' histograms. This value could be the average or the highest score of the remaining docIDs a peer has not sent to P_{init}, or some alternative score. For example, after histogram-based statistical analysis, the average score augmented by a multiple of the standard deviation adequate to capture a certain percentile of the remaining score distributions could be used. Obviously, this is beyond the scope of this work. However, we present an approach that is based on the above insight avoids the conundrum of selecting an appropriate R value.

The basic idea is for peers in the third step of the algorithm to construct CFs that are no longer simple bit maps: a non-zero value in a CF position indicates now the cell number of the docID hashing into this position.

Specifically, in the third step of KLEE:

1. For each docID that belongs into its candidate list, each peer hashes the docID and stores, in the CF position indicated by the hash, the cell number of the peer's histogram into which this docID belongs. Formally, $CF[i] = r$, if and only if $hash(docID) = i$, and $lb[r] = score(docID) = ub[r]$.

2. P_{init} after receiving the peer CFs constructs as before the $m \times b$ matrix CFM.

3. Finally, P_{init} defines a column of CFM, j, $1 \leq j \leq b$, as interesting if and only if:

$$\sum_{i=1}^{m} ub_i[CFM[i,j]] > min\text{-}k$$

where $ubi[r]$ represents the upper bound of cell r in the histogram of peer P_i. Note that by using the upper bound score of the cell to which a $docID$ belongs, the definition of interesting CFM columns ensures that no $docID$ that could attain a TotalScore higher than min-k would be missed.

Obviously, the new definition of the interesting columns of the CFM structure automatically brings about a new definition of the peers' candidate lists to be retrieved in the final step of KLEE.

The new method for selecting interesting columns introduces bandwidth savings and improves the quality of the expected result top-k list. However, note that these benefits come at the expense of using additional bits for the contents of CFs. Since cell numbers are stored now in CFs, a number of bits equal to $log_2(n)$, where n is the number of histogram cells, are required. Since n is typically fairly small (e.g., = 100), this cost is still small.

Note that instead of using the upper bound values of cells, the average or even the lower bounds could be used, offering trade-offs with respect to higher bandwidth savings versus reduced accuracy of the resulting top-k list.

4.4 Experimentation

4.4.1 Experimental Setup

Our implementation of the testbed and the related algorithms was written in Java. All peer related data were stored locally at the peer's disk. Experiments were performed on 3GHz Pentium machines. For simplicity, all processes ran on the same server. Real-World Data Collections and Queries. Two real-world data collections were used in our experiments: GOV and IMDB. The queries for the former contained text attributes, whereas queries for the latter collection contained text and structured attributes.

- The **GOV collection** consists of the data of the TREC-12 Web Track and contains roughly 1.25 million (mostly HTML and PDF) documents obtained from a crawl of the .gov Internet domain (with total index list size of 8 GB). The original 50 queries from the Web Track's distillation task were used. These are term queries, with each query containing up to 5 terms. In our experiments, the index lists associated with the terms contained the original document scores computed as $tf * logidf$. tf and idf were normalized by the maximum tf value of each document and the maximum idf value in the corpus, respectively.

- In addition, we employed an **extended GOV (XGOV)** setup, which we utilized to test the algorithms' performance on a larger number of query terms and associated index lists. The original 50 queries were expanded

by adding new terms from synonyms and glosses taken from the WordNet thesaurus[2]. The expansion resulted in queries with, on average, twice as many terms, with the longest query containing 18 terms.

- **The IMDB collection** consists of data from the Internet Movie Database (http://www.imdb.com). In total, our test collection contains about 375,000 movies and over 1,200,000 persons (with a total index list size of 140 MB), structured into the object-relational table schema Movies (Title, Genre, Actors, Description). Title and Description are text attributes and Genre and Actors are set-valued attributes. Genre contains 2 or 3 genres. Actors included only those actors that appeared in at least 5 movies. For similarity scores among Genre values and among Actors we precomputed the Dice coefficient for each pair of Genre values and for each pair of actors that appeared together in at least 5 movies. So the similarity for genres or actors x and y is set to

$$s(x,y) = \frac{2(\#movies\ containing\ x\ and\ y)}{\#movies\ with\ x + \#movies\ with\ y}$$

, and the index list for x contains entries for similar values y, too.

- **Synthetic Data Collections and Queries**. Our synthetic benchmarks allow the evaluation of the algorithms under different input data characteristics. We systematically study the effect of (i) the skewness in score distributions and (ii) of the correlation among queried terms on the algorithms' performance. We created index lists having score distributions following the Zipf law [Zip49], varying the Zipf parameter (θ), to create varying skewness. For each set of real-world collections (e.g. GOV and XGOV) we kept the docIDs in the original index lists in tact and simply replaced the scores to follow a Zipf distribution with values of $\theta = 0.3$, 0.7, and 1.0. The set of queries was the same as in the corresponding GOV and XGOV benchmarks. We coined these synthetic benchmarks **Zipf-GOV** and **Zipf-XGOV**. Finally, in real-world applications there will often be correlations among the query terms. To systematically test this, we generated synthetic index lists that had controlled overlap among their docIDs, using a parameter O. Given any index list $I(t_1)$ its overlap with another $I(t_2)$ was created as follows: for each of the top-k docIDs in $I(t_1)$, a random (uniform) value, v, was selected in the range $[k+1, O]$ and this docID was inserted in I(t2) at position v. By controlling the value of O between $[k+1, sizeof(I(t_2))]$, we create stronger or weaker correlations (for smaller or greater values of O, respectively). We created 10 such index lists. The queries in these Overlap benchmarks were queries involving t terms, $t = 2, ..., 10$, with each query selecting randomly t index lists from the set of 10.

[2] `http://www.cogsci.princeton.edu/~wn/`

4.4.2 Tested Algorithms

DTA: This is a Distributed TA algorithm, an extension of the standard TA algorithm. Each peer partitions its sorted index list into batches, with each batch having k entries. DTA proceeds in phases, in each phase each peer sends its next batch. After each phase, the coordinator runs the TA algorithm on the collected entries and stops when all uncollected index entries can be pruned away.

TPUT: This is the 3-phase algorithm as described in [CW04]. TPUT comes in two flavors: the original and a version with compression for long docIDs. This optimized version instead of sending (docID, score) pairs, hashes the docID into a hash array where it stores its score and sends the hash array of scores. Even in the experiments conducted in [CW04] the compressed optimized version did not always perform better. Furthermore, KLEE could also use compression for the filters in Step 1 and the sparse CFs in step 3. For these reasons, we report only the results for the original TPUT version.

X-TPUT: As one of our key contributions is to show the suitability and significant benefits of approximate top-k algorithms, we implemented a new version of TPUT, which we coined X-TPUT. X-TPUT essentially consists of only the first two phases of TPUT. We tested X-TPUT given our expectation that even with some missing scores, which TPUT retrieves in the 3rd phase, it should still be possible to develop an algorithm that performs much better than TPUT, at a small precision penalty.

KLEE-3: This is KLEE with only three steps, two communication phases - i.e., the version of KLEE without Step 3, the Candidate List Reduction Step.

KLEE-4: This is KLEE with all four steps, three communication phases engaged.

4.4.3 Performance Metrics

Cost: Bandwidth. This represents the total number of bytes transferred between the query initiator and the cohort peers. This is our primary metric, since it is widely regarded to be critical in the envisioned applications. Cost: Query Response Time. This represents the elapsed, "wall-clock" time for running the benchmarks. Quality: Relative Recall. This represents the fraction of the top-k results produced that are in the "true" top-k results without any approximations. By construction, DTA and TPUT have a recall value of 1. Quality: Normalized Score Error. The score error is the average of the differences between the score of the i-th position in an algorithm's result top-k list and the score in the i-th position in the "true" top-k result, for all $1 \leq i \leq k$. By, construction, DTA and TPUT have a score error value of 0. Note that this is an important metric since the recall value alone may lead to erroneous conclusions. As an extreme example, in cases where the top-$2k$ docIDs have very small score differences, it is possible that a top-k result list can have recall close to 0, while being a very good result with only negligible score differences from the true top-k result. Since the score error may be a very small number, we normalize it by dividing

it with the *min-k*. We also computed the footrule distance for the ranks of approximate vs. exact top-k results.

4.4.4 Experimental Results

We report on experiments performed for each of the benchmarks, GOV, XGOV, IMDB, Zipf-GOV, Zipf-XGOV, and Overlap. In all experiments queries are for the top-20 results. KLEE algorithms assume that peers in the first step send to the query initiator filters for enough histogram cells, whose cumulative score is a certain percentage (e.g., 5%, 10%, or 20%) of the total score mass. In the experiments we use $c = 10\%$. In KLEE, the Bloom filters were configured as follows: For the 1st step, the filters for each cell of a peer's histogram were long enough to ensure that $pfp < 0.004$. This creates sparse filters, but helps to avoid overestimating the *min-k* due to false positives. For the 3rd step, the size of peers' CFs ensured that $pfp < 0.06$. This larger pfp is deemed as an appropriate compromise between unnecessarily long filters versus a few (6%) more (docID, score) pairs that need be sent (for docIDs that were mistakenly assumed to be in the interesting columns of the CFs of peers). Running the experiments over multiple nodes in a network would be inherently vulnerable to interference from other processes running concurrently and competing for cpu cycles, disk arms, and network bandwidth. To avoid this and produce reproducible and comparable results for algorithms ran at different times, we opted for simulating disk IO latency and network latency which are dominant factors. Specifically, each random disk IO was modeled to incur a disk seek and rotational latency of 9 ms, plus a transfer delay dictated by a transfer rate of 8MB/s. For network latency we utilized typical round trip times (RTTs) of packets and transfer rates achieved for larger data transfers between widely distributed entities [SL00]. We assumed a packet size of 1KB with a RTT of 150 ms and used it to measure the latency of communication phases for data transfer sizes in each connection up to 1KB. When cohorts sent more data, the additional latency was dictated by a "large" data transfer rate of 800 Kb/s. This figure is the average throughput value measured (using one stream – one cpu machines) in experiments conducted for measuring wide area network throughput (sending 20MB files between SLAC nodes (Stanford's Linear Accelerator Centre) and nodes in Lyon France [SL00] using NLANR's iPerf tool [Tir03]. Hence, the overall response times were the sum of cpu times for an algorithm's local processing, IO times, and network communication times. Since cohorts are running in parallel, the longest time was considered in each phase.

4.4.5 Performance Results

On Synthetic Benchmarks

Bandwidth Costs. Figure 4.4 shows the bandwidth results for Overlap. We show results for $\theta = 0.7$, and $t = 5$-term queries. Ω was varied to correspond to the index list positions capturing from 10% to 100% of the total score mass.

We see that the KLEE algorithms show excellent performance. KLEE-4 outperforms the TPUT algorithms by a factor ranging from approximately 2.5 to more than an order of magnitude. Intuitively, higher correlations imply that the HistogramBlooms have a greater chance to work: when calculating the TotalScores of docIDs in the first phase, any missing scores will be (with high probability) found in the filters for the docIDs in the top histogram cells sent by peers. This results in much better approximations of $min\text{-}k$, which in turn results in not having to go very deep into the peer index lists in the subsequent phases to retrieve candidates.

The difference in the performance between KLEE-3 and KLEE-4 shows the benefits introduced by the CFM filtering in the 2nd communication phase of KLEE-4. KLEE-3 also enjoys much better performance, especially for higher term correlations. As Ω values increased, the performance gains of KLEE-3 vs TPUT and X-TPUT decreased, due to the inability of HistogramBlooms to significantly help. Perhaps surprisingly, DTA performs well, for queries with higher overlap, since a high overlap implies that, after a relative small number of batches, DTA has gone deep enough in all index lists. (However, as we shall see later, this comes at a very high cost in response times). Figures 4.5 and 4.6 show the bandwidth results for Zipf-GOV and Zipf-XGOV, respectively, for $\theta = 0.7$ (similar results occur with all other values of θ). In all cases, the KLEE algorithms outperform the TPUT competitors. In particular, for Zipf-GOV and Zipf-XGOV, KLEE-4 wins by a factor of 2, compared to TPUT and X-TPUT.

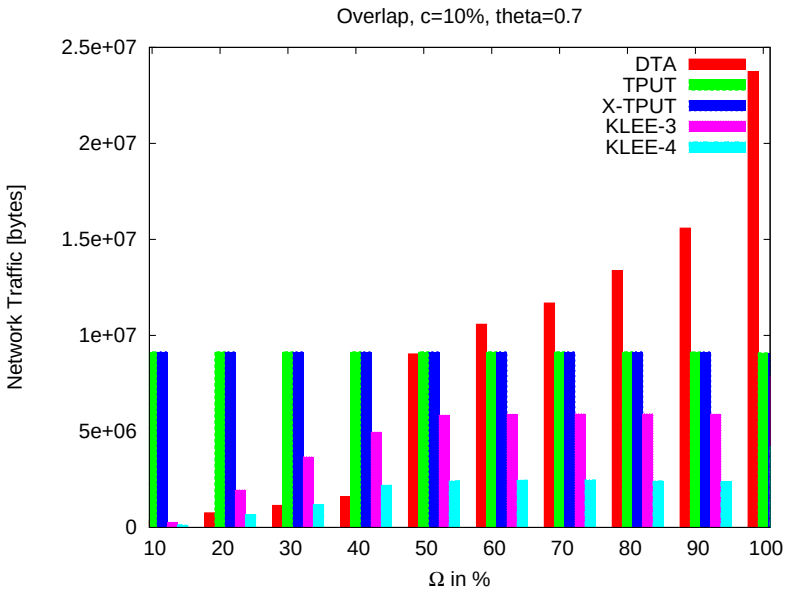

Figure 4.4: Bandwidth for the Overlap Benchmark ($\theta = 0.7$, $c = 10\%$)

DTA performs very well for a small number of terms/peers. For larger numbers of terms/peers, DTA's bandwidth performance deteriorates, and for more than ten terms it is consistently and by far the worst performer. With respect to the TPUT algorithms vs KLEE-3, we note that for queries with more than 3 terms/peers, KLEE-3 outperforms X-TPUT, by about 10% to about 50%. These smaller gains of KLEE-3 are attributable to the very small term correlations in these benchmarks. Finally, in general, for less skewed score distributions, as shown here, X-TPUT and TPUT have similar bandwidth performance. Intuitively, this is due to TPUT and X-TPUT using the same score threshold value. The less skewed a score distribution is, the larger number of docIDs (having higher scores than the threshold) are sent by each peer to the coordinator. Thus, the smallest is the missing information at the coordinator, which is retrieved by TPUT in the 3rd phase. Tables 4.1, 4.2 and 4.3 present the aggregate picture for most metrics we used, for the Overlap, Zipf-GOV, and Zipf-XGOV benchmarks. In total bandwidth, KLEE-4 is better than both TPUT algorithms by a factor of about 8 in Overlap and by more than 2 in Zipf-GOV and Zipf-XGOV. KLEE-3 is better by a factor of about 2.5 in Overlap and by about 10% in the other two.

Response Times. We see a similar picture from Tables 4.1, 4.2 and 4.3, which show total benchmark times (i.e., for the entire batch of 50 queries). In Table 4.1, for the Overlap benchmark, KLEE-4 (KLEE-3) is shown to outperform the TPUT algorithms by a factor better than 4 (2). Similarly, for the Zipf-XGOV benchmark, KLEE-4 (KLEE-3) outperforms X-TPUT and TPUT by a factor higher than 4 (25%). For Zipf-GOV, KLEE-4 is better by about 2.5 (3.5) times than X-TPUT (TPUT), respectively. The DTA times are very disappointing, due to very high number of random IOs. Overall, KLEE-4's, response times are better by 1-2 orders of magnitude.

Result Quality. Tables 4.1, 4.2 and 4.3 also depict results using different metrics for result quality, namely: relative recall, normalized average score error, and average rank distance. With average recall being higher than 90%, and very small rank distance and score errors, the approximate algorithms, and especially KLEE, prove themselves as the algorithms of choice, given their great performance.

On Real-World Benchmarks

Bandwidth Costs. Figures 4.7 and 4.7 and the first columns of Tables 4.7, 4.8, and 4.9 show the bandwidth results for GOV, XGOV, and IMDB respectively. Figure 4.8 shows bandwidth consumption for IMDB. We observe that, again, KLEE-4 is the strongest performer, outperforming X-TPUT by a factor of about 2 (for > 2 terms) in GOV, by a factor of between 2 and 3 in XGOV, and by a factor of about 3 for IMDB. Against TPUT, KLEE-4 is better by a factor of up to 6 in GOV and by up to more than an order of magnitude in XGOV, and by similar factors for IMDB. KLEE-3 and X-TPUT performed comparably. X-TPUT outperforms KLEE-3 by better than 20% in GOV, while KLEE-3 wins by more than 15% in XGOV.

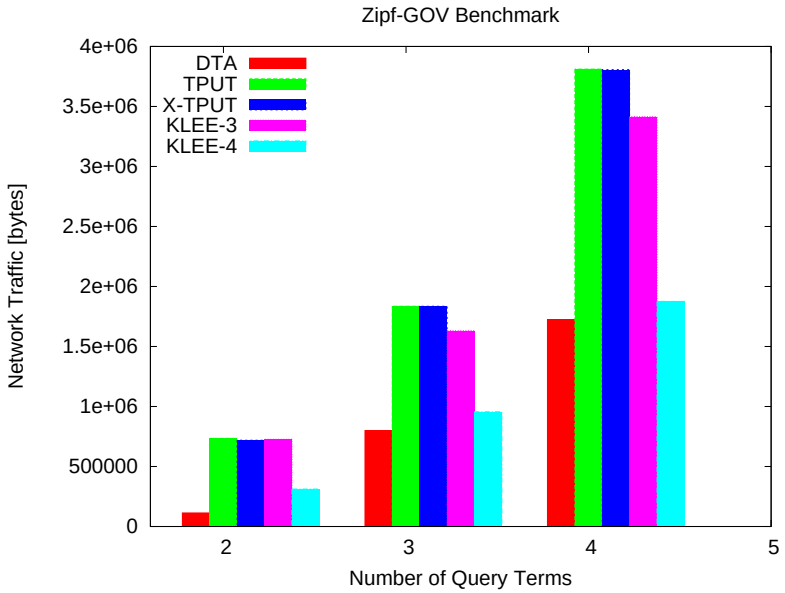

Figure 4.5: Bandwidth for the Zipf-GOV Benchmark

It is interesting to note that X-TPUT in these benchmarks outperforms TPUT. Since index lists are very skewed, the score threshold of $min\text{-}k/m$ points to a depth in the index lists which is not surpassed by a large number of docIDs.

Thus, unlike the synthetic benchmarks reported, there is a large mass of information that TPUT must retrieve in the third phase, which explains the better performance of X-TPUT. However, note from Figures 4.7, 4.8, and 4.9 that as the number of terms/peers increases, both TPUT and X-TPUT start performing worse (with KLEE-3 consistently surpassing X-TPUT, for example). Finally, again, DTA is in general performing very poorly except for very small numbers of terms. Response Times. The same trends are noted for response times. Both KLEE algorithms significantly outperform TPUT and DTA. X-TPUT approaches the response times of KLEE for smaller-term queries, (e.g. in GOV) but as the number of terms increases it becomes worse by a factor of about 2 (e.g. in XGOV). The KLEE algorithms are also best in terms of fewer random and sequential local IOs at peers. This shows that KLEE incurs the lightest local peer work.

Result Quality. Tables 4.4, 4.5 and 4.6 show that all approximate algorithms continue to provide acceptable result quality. Average recall values for KLEE-4 (KLEE-3) are at 90% (90%) and 79% (83%) for GOV and XGOV respectively and average score errors are about 2% and 5% of $min\text{-}k$. In light of KLEE's strong performance, this is definitely acceptable.

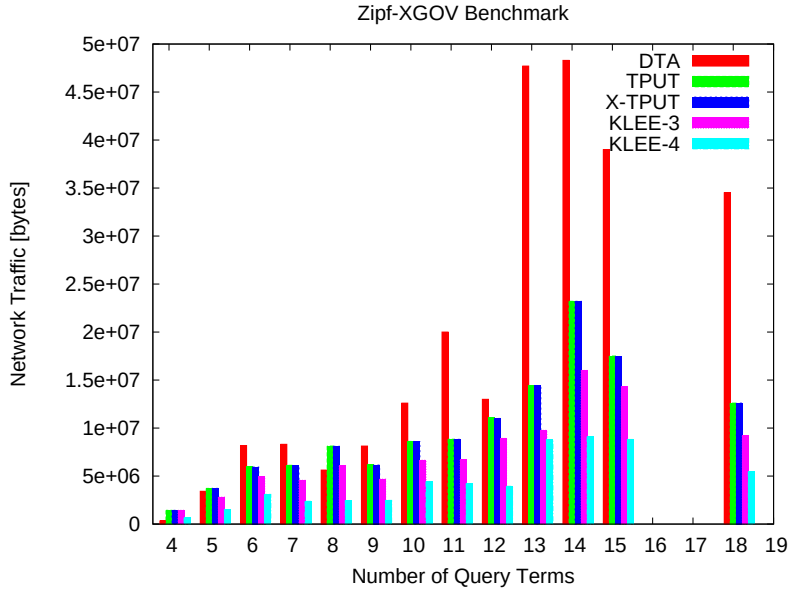

Figure 4.6: Bandwidth for the Zipf-XGOV Benchmark

Overlap W=30%	bytes	time in ms	recall	error/mink	rank distance	sorted	random
TA	229,264	31,484	1	0	0	1,612	30
TPUT	1,830,181	5,854	1	0	0	14,173	0
X-TPUT	1,830,181	5,667	1	0	0	14,173	0
KLEE 3	735,756	2,594	0.92	0.0003	1.45	5,560	0
KLEE 4	238,541	1,309	0.91	0.0003	1.39	5,553	0

Table 4.1: Performance Results for the Overlap Benchmark

GOV c=10%	bytes	time in ms	recall	error/mink	rank distance	sorted	random
DTA	17,752,769	3,532,180	1	0	0	89,241	133,338
TPUT	53,494,903	576,713	1	0	0	1,262,745	15,998
X-TPUT	53,011,252	404,991	0.99	0.001	0.13	1,262,745	0
KLEE 3	49,861,342	367,931	0.97	0.002	0.87	1,182,434	0
KLEE 4	25,057,920	160,585	0.94	0.004	1.04	1,182,434	0

Table 4.2: Performance Results for the Zipf-GOV Benchmark ($\theta = 0.7$)

XGOV c=10%	bytes	time in ms	recall	error/mink	rank distance	sorted	random
DTA	617,009,260	39,582,682	1	0	0	443,040	2,486,650
TPUT	377,928,880	1,599,581	1	0	0	5,057,570	6,465
X-TPUT	377,097,644	1,521,220	0.98	0.002	0.36	5,057,570	0
KLEE 3	287,294,812	1,189,891	0.91	0.012	1.70	3,908,467	0
KLEE 4	165,077,807	375,077	0.92	0.011	1.43	3,924,437	0

Table 4.3: Performance Results for the Zipf-XGOV Benchmark ($\theta = 0.7$)

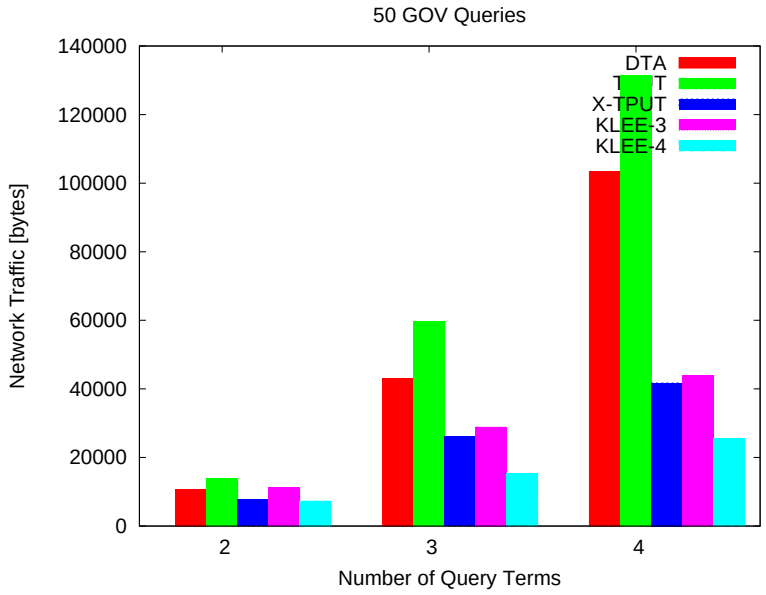

Figure 4.7: Bandwidth for the GOV Benchmark

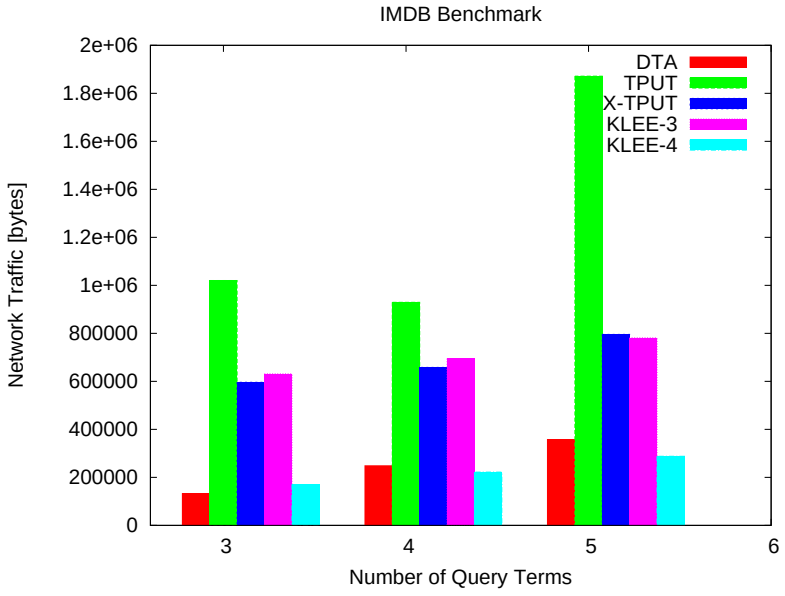

Figure 4.8: Bandwidth for the IMDB Benchmark

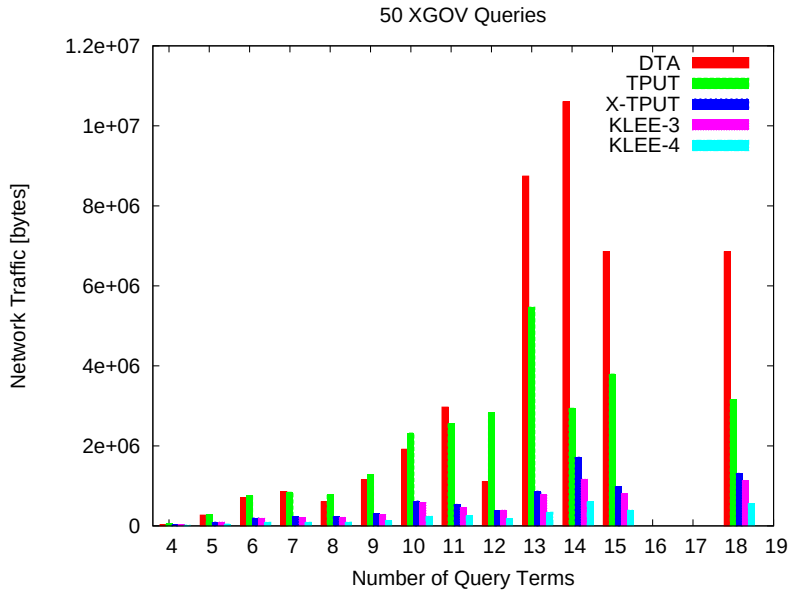

Figure 4.9: Bandwidth for the XGOV Benchmark

GOV c=10%	bytes	time in ms	recall	error/mink	rank distance	sorted	random
DTA	1,172,446	190,259	1	0	0	6,043	8,229
TPUT	1,505,290	185,049	1	0	0	13,180	13,754
X-TPUT	597,991	31,432	0.89	0.026	1.21	13,180	0
KLEE 3	722,664	28,319	0.90	0.018	1.16	11,652	0
KLEE 4	440,868	39,564	0.90	0.022	1.27	11,652	0

Table 4.4: Performance Results for the GOV Benchmark

XGOV c=10%	bytes	time in ms	recall	error/mink	rank distance	sorted	random
DTA	92,587,264	3,740,677	1	0	0	40,940	289,468
TPUT	70,044,884	2,346,882	1	0	0	235,809	213,906
X-TPUT	19,236,084	96,153	0.91	0.027	1.12	235,809	0
KLEE 3	16,690,912	88,271	0.83	0.046	2.91	203,174	0
KLEE 4	7,920,774	56,609	0.79	0.052	3.25	203,174	0

Table 4.5: Performance Results for the XGOV Benchmark

IMDB c=10%	bytes	time in ms	recall	error/mink	rank distance	sorted	random
DTA	3,182,737	581,226	1	0	0	16,110	28,836
TPUT	16,152,355	1,148,847	1	0	0	282,013	9,708
X-TPUT	8,406,897	92,137	0.73	0.026	3.85	282,013	0
KLEE 3	8,592,431	92,745	0.70	0.026	4.14	276,795	0
KLEE 4	2,845,225	33,616	0.69	0.027	4.33	276,795	0

Table 4.6: Performance Results for the IMDB Benchmark

Chapter 5

Statistical Estimators and Automatic Parameter Tuning

5.1 Modeling Score Distributions

In the following we will shortly discuss the statistics that we use as a basis for our cost model. The gathering of these distributed statistics is then part of the first phase of our algorithm: The query initiator retrieves from each peer the top k documents along with the statistics for the particular attribute (i.e. term).

The main difficulty here is, however, to precisely describe single (per-attribute) score distributions in a way that allows for a highly accurate prediction of the number of documents with score above a certain threshold. Moreover, as we are interested in employing a hierarchical top-k algorithm, and thus during optimization logically split the top-k query into several sub-queries we additionally need score distribution models for aggregated data, as we will see below.

For text-based IR with keyword queries, the query-to-document similarity function is typically based on statistics about frequencies of term occurrences, e.g., the family of tf*idf scoring functions [Cha02] or more advanced statistical language models [CL03]. Here, *terms* are canonical representations of words (e.g., in stemmed form) or other text features.

5.1.1 Poisson Distributions

Given an index list for a particular term we can model the frequency of occurrences using a Poisson Distribution:

$$Pr_P(x, \theta) = \frac{e^{-\theta} * \theta^x}{x!}$$

$\pi(k, \theta)$ is the probability that a particular term occurs exactly k times in a particular document. The parameter θ is the mean (and the variance) of the distribution and is used to adapt the Poisson model to a given distribution.

One important technique that we need to apply here is the convolution of two distributions in order to obtain a model for the joint distribution.

If X and Y are independent discrete random variables, each taking on the values $0, 1, 2, 3, ...$ then Z takes on the values $k = i + j$ $(i, j = 0, 1, 2, 3, ...)$ and

$$P[Z = k] = \sum_{k=i+j} p_X(i)p_Y(j) = \sum_{i=0}^{k} p_X(i)p_Y(k-i). \tag{5.1}$$

The convolution of two Poisson distributions p_1 and p_2 is [All90]

$$\begin{aligned} p(x) &= \sum_{i=0}^{x} p_1(i)p_2(x-i) \\ &\sum_{i=0}^{x} \frac{e^{-\theta_1}\theta_1^i}{i!} \frac{e^{-\theta_2}\theta_2^{x-i}}{(x-i)!} \\ &= \frac{e^{-(\theta_1+\theta_2)}}{x!} \sum_{i=0}^{x} \frac{x!\theta_1^i\theta_2^{k-i}}{i!(x-i)!} \\ &= \frac{e^{-(\theta_1+\theta_2)}}{x!} \sum_{i=0}^{x} \binom{x}{i} \theta_1^i\theta_2^{x-i} \\ &\underline{= e^{-(\theta_1+\theta_2)}(\theta_1+\theta_2)^x} \\ &\qquad x! \end{aligned}$$

We see that the convolution of a Poisson distribution with parameter θ_1 and a Poisson distribution with parameter θ_2 is a Poisson distribution with parameter $\theta_1 + \theta_2$. No other probabilistic distribution has this property that the convolution reproduces the same distribution function just with different parameters. Figure 5.2 shows the "convolution" of two Poisson distributions where the aggregation function is the maximum, the simple summation, and the weighted summation.

Unfortunately, simple Poisson distributions are not a particularly good fit for capturing the scores of real data. However, mixtures of Poisson distributions are a fairly accurate, realistic model [CG95]. Figure 5.1 shows examples of Poisson distributions and Poisson mixture distributions.

Two-Poisson Model

The Two Poisson Model is a simple example of a Poisson mixture.

$$Pr_{2P}(x, \theta) = \alpha Pr_P(x, \theta_1) + (1 - \alpha)Pr_P(x, \theta_2)$$

Harter [Har75] showed how to use the method of moments to fit the three parameters of the Two-Poisson model θ_1, θ_2, and α, from the first three moments

of observation. Let R_i be the i^{th} moment around zero. It can be estimated empirically by from observed values $R_i \approx \sum_k k^i Pr_E(k)$. $\alpha = \frac{R_1 - \theta_2}{\theta_1 - \theta_2}$. θ_1 and θ_2 are the roots of the quadratic equation: $a\theta^2 + b\theta + c = 0$, where:

$$a = R_1^2 + R_1 - R_2$$

$$b = R_1^2 - R_1 R_2 + 2R_2 - 3R_2 + R3$$

$$c = R_2^2 - R_1^2 + R_1 R_2 - R_1 R_3$$

Mixtures of Poisson distributions are a fairly accurate, realistic model [CG95]. Note that each Two Poisson Model requires only 3 floating point numbers so that the additional network resource consumption is negligible.

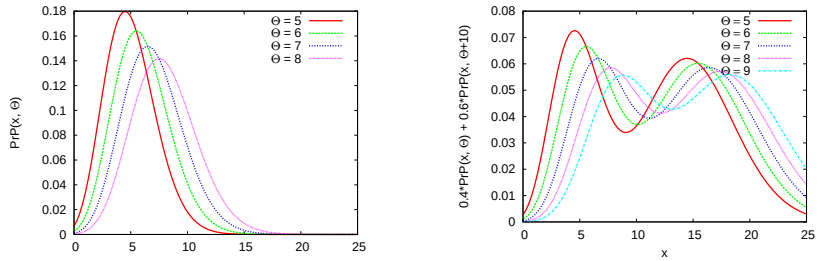

Figure 5.1: Examples for Poisson and Poisson Mixture Distributions

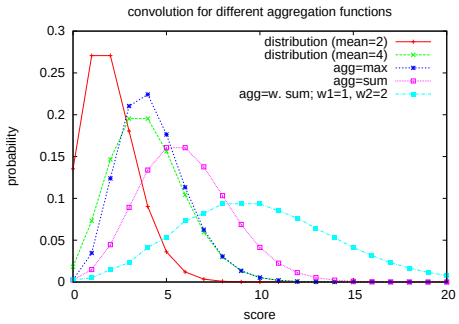

Figure 5.2: Convolutions for different aggregation functions

5.2 Cost Prediction Model

Consider a top-k query over input lists L_i, $i = 1...m$, spread across m peers. We are interested in the following three prediction tasks:

1. Given a scan depth n' on all lists, i.e., retrieving the locally top-n' items from each list and aggregating them, predict the value of *min-k*, i.e., the aggregated value of the rank-k item in the global ranking (based on the given information, which is typically partial for individual items).

2. Given a threshold value θ for the aggregated values in the global ranking, predict a scan depth n' that we need to use on all lists in order to ensure, with high probability, that we obtain all items with aggregated value above θ.

3. Predict the network bandwidth consumption of these query steps, i.e., the number of list entries that need to be sent across the network.

In this section, we mostly focus on the first task as it is the most difficult one. The second task can be reduced to this first one, by binary search over n' (but there are also other ways of implementing it). Finally, the third task uses the results of the other two tasks in a straightforward manner: we merely need to know the number of items that are fetched from each peer in order to calculate the bandwidth consumption, which we assume to be the most important measure of networking costs. Note that the number of items fetched per input list is also a direct measure of the local execution cost at each peer.

In the following, we address two different situations in which we need a cost prediction model. If we only know that our queries involve m peers but cannot tell which individual peers these will be, we need to assume the same local value distribution function. This situation is given at system configuration time or when we employ sampling over peers (see Section 6.3); it requires a relatively simple distribution model with a robust way of parameter estimation. On the other hand, when a concrete query is issued, we know which individual peers are involved and can utilize more specific knowledge about the local value distributions of these peers (e.g., histograms). Obviously, as inferring predictions from these distributions is then part of the query response time, it needs to be computationally light-weight.

5.2.1 Value Distributions

We first address the situation where we only know that m peers are involved, but have no information about which ones and thus no specific value distributions. Thus we consider the same distribution for all peers, and would like to characterize it by few parameters. Analyses of real data such as text corpora or query logs show that Poisson mixes are good approximations of the, usually skewed, value distributions [CG95]. Zipf or Pareto distributions (i.e., power-law distributions) would be obvious candidates, too, but they have only one parameter and are therefore cruder approximations, and they are mathematically less tractable than Poisson mixes.

For the Poisson-mix model we assume, without loss of generality, that each

peer has z equidistant values val_j, ranging from $val_1 = 1/z$ to $val_z = 1$. Then

$$f_{Si}(x) = P[Si = val_j] = \alpha e^{-\beta}\frac{\beta^j}{j!} + (1-\alpha)e^{-\gamma}\frac{\gamma^j}{j!}$$

with parameters α,β,γ that need to be (and can easily be) fit to the value distributions of the real data. The corresponding cumulative distribution function is denoted $F_{Si}(x)$; for Poisson distributions or mixes there is no closed analytic representation, but it can be written using the incomplete Gamma function, which in turn can be numerically computed [PFTV88].

When scanning all m input lists up to scan depth n', we see a randomly drawn item d in the top-n' entries of q out of m lists with probability

$$P_{seen}(q) = \binom{m}{q}(\frac{n'}{n})^q(\frac{n-n'}{n})^{m-q}$$

with expectation

$$E_{seen} = mn'/n =: m'$$

. The probability that we see item d in at least one list is

$$1 - (1 - \frac{n'}{n})^m$$

, and the expected number of distinct items seen is

$$E_{dist} = 1 - (1 - \frac{n'}{n})^m n =: n''$$

. For each of these n'' items we want to characterize its total value that results from the aggregation over the m' lists in which we have, on average, seen an item. We denote these aggregated values by the random variable S. The distribution of S, $f_S(x)$, is obtained by the m'-fold convolution of $f_{Si}(x)$, the distribution in each list. This assumes stochastic independence between different lists, a postulate commonly made for tractability. Although the assumption rarely holds in practice, models based on independence have been very successful for many prediction tasks and applications that require such statistical reasoning.

The convolution of Poisson mixes yields again a Poisson-mixture distribution with $m' + 1$ mixture components [All90]. The generating function

$$G(z) = E[z^x] = \sum_{x=0}^{\infty} f(x)z^x$$

for the Poisson distribution with parameter γ is

$$G_{S_i} = e^{\gamma(z-1)}$$

. The generating function of a linear combination $\alpha f(x) + (1-\alpha)g(x)$ is the linear combination of the two components, and the generating function for a convolution of distributions is the product of the underlying generating functions [All90]. This way we can derive that

$$G_S(z) = (\alpha e^{\beta(z-1)} + (1-\alpha)e^{\gamma(z-1)})^{m'} = \sum_{\nu=0}^{m'}\binom{m'}{\nu}\alpha^{\nu}(1-\alpha)^{1-\nu}e^{\nu\beta+(m'-\nu)y)(z-1)}$$

for the aggregated value distribution $f_S(x)$, and we see that

$$f_S(x) = P[S = val_j] = \sum_{\nu=0}^{m'} \binom{m'}{\nu} \alpha^{\nu} (1-\alpha)^{1-\nu} e^{-(\nu\beta+(m'-\nu)\gamma)} \frac{(\nu\beta+(m'-\nu)\gamma)^j}{j!}$$

, i.e. a mixture of $m'+1$ Poisson distributions with specific parameters, for $j = 1...m'z$ and $val_j = j/z$ (i.e., covering aggregated values from 0 to m'). We show in Subsection 5.2.2 how to utilize this distribution model for aggregated values in order to predict *min-k* for given scan depths on m lists.

In query-time situations where we know which individual lists are involved, we can utilize a more detailed model with precomputed list-specific statistics. Instead of the more commonly used equi-width or equi-depth histograms for density functions, we use a more accurate model with *linear spline histograms* to approximate cumulative distribution functions (cdf) of local values. As cdf's are monotonically increasing, we can leverage efficient algorithms for function approximation. Linear splines can be constructed such that a (low) maximum error can be guaranteed at every point. The problem of minimizing the maximum error has been well studied in the area of computational geometry [Goo95], but the required algorithms are computational complex. Instead, we use a "taut string" approximation technique [Pow98] that constructs the linear spline in linear time.

Linear splines cannot be convolved directly, as different splines may have different interpolation points. Instead, we compute the convolution by using histograms as an intermediate representation. For two histograms with different cell boundaries, we determine all cell intersections and then perform the convolution on the resulting finer-grained but now compatible histograms. The resulting histogram has more cells than the original input and thus would require more memory, but as the spline construction can be done in linear time, we can quickly rebuild a spline representation for the convolved distribution. In our implementation we used splines with 100 interpolation points.

5.2.2 Estimating *min-k*

For estimating *min-k*, we now operate on the convolved distributions analyzed above. Each of the expected n'' items that we see with scan depth n' has an aggregated value according to the probability density function $f_S(x)$. Denote these random variables by $S_1, ..., S_{n''}$ and order them in ascending order. Without loss of generality we can renumber them such that $S_1 \leq S_2 \leq ...S_{n''}$. We are interested in the value of the rank-k item, namely $S_{n''-k+1}$. This estimation problem is a standard problem in order statistics [DN03]. $S_{n''-k+1}$, the rank-k order statistics, is itself a random variable, which is difficult to characterize in its full distribution. But we are only interested in its expectation $E[S_{n''-k+1}]$. A first-order approximation to this is the $((n''-k+1)/n'')$-quantile of $F_S(x)$; more accurate approximations based on a Taylor-series expansion are derived [DN03] but are difficult to compute (including evaluating derivatives of the quantile

function). We will use the simpler first-order approximation

$$E[S_{n'-k+1}] \approx F_S^{-1}((n'' - k + 1)/n'')$$

where F_S^{-1} denotes the quantile function.

Given the Poisson-mix or linear spline representations, we can apply the above computation to either one of the two representations. For both models, calculating the quantile can simply be done by binary search; there is no need (and probably also no easy way) for an analytic solution. Our implementation is very efficient.

Figure 5.3 shows the average relative error in the *min-k* estimation, i.e. $|estimated\text{-}min\text{-}k - true\text{-}min\text{-}k|/true\text{-}min\text{-}k$.

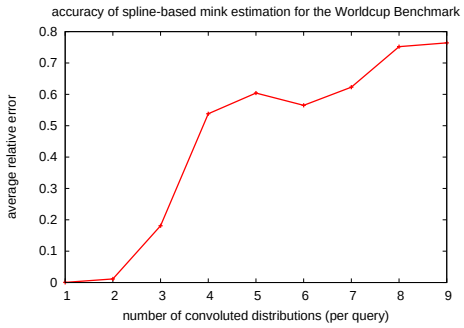

Figure 5.3: The average relative error in the *min-k* estimation

Chapter 6

The GRASS Algorithms

This Chapter presents the family of GRASS[1] algorithms, that employ three different optimization techniques. The core of the GRASS algorithms is the previously presented KLEE framework, however, the developed optimization techniques can be applied to all KLEE related algorithms as well, such as TPUT.

To scale up top-k query processing to hundreds of nodes, this chapter contributes two novel techniques:

1. The flexible formation of hierarchical groups of node subsets that are considered together. This divide-and-conquer paradigm, illustrated in Figure 6.1 (b), avoids overly broad top-k aggregation queries that involve too many nodes at the same time and could lead to (incoming) bandwidth bottlenecks at the root of the aggregation. On the other hand, it introduces the combinatorial problem of choosing appropriate groups and forming a tree of cascaded top-k operators (possibly with different k at different stages). We provide both exact methods and heuristic approximations for solving this optimization. The idea is based on our own work in [NM07].

2. While previous methods have usually propagated uniform scan depth thresholds to other peers, we propose an adaptive method for choosing different scan depth thresholds at different nodes, driven by the statistical information about the value distributions in the local lists (cf. Figure 6.1 (c)).

For additional scaling, with queries possibly running over thousands of nodes, we contribute a third technique:

3. Choosing a sufficiently small subset of nodes as samples, based on a statistical error estimation (cf. Figure 6.1 (d)). The sample contains nodes that are most likely to contribute the highest values to the top-k aggregation.

[1] Grass the last name of Günther Grass, a Nobel Prize-winning German author.

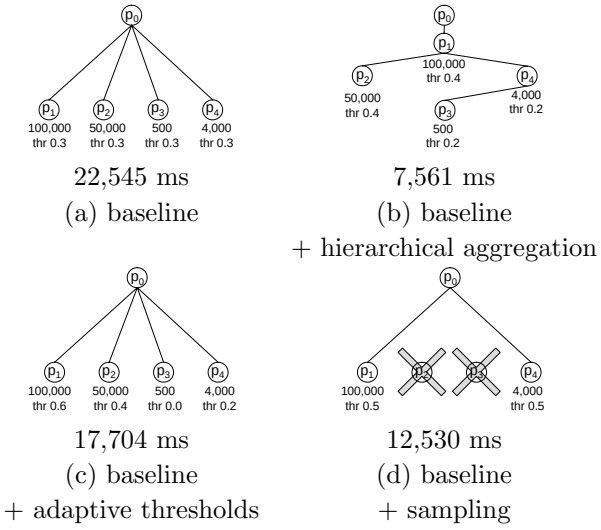

Figure 6.1: Execution plans illustrating the optimization techniques

Depending on the estimated error, the sample can optionally be increased in an additional round, or a small number of top-k candidate items may be probed at all network nodes.

All three techniques are based on the statistical cost predictor that is described in Chapter 5 to estimate the cost of the considered groupings, scan depths, or samples.

A standard way of performing distributed top-k aggregation queries is illustrated in Figure 6.1 (a). The figure shows 4 input lists on 4 different peers and the message flow to a 5th peer that has posed a top-k query. The 4 lists have different sizes, and we assume that the query processing uses a uniform value threshold of 0.3 for its scan depth. We will later contrast this execution plan with better ones based on our methods. Figure 6.1 also shows response times measured in our testbed, as anecdotic evidence of our performance gains.

6.1 Adaptive Thresholds

After determining an initial *min-k* threshold, the second phase of TPUT and KLEE request all data items that can may possibly qualify for the top-k results. A conservative way of doing this is the TPUT method which distributes the necessary value mass evenly over all input lists and thus requests all items with a local value above *min-k*$/m$. However, this uniform threshold for all lists is only one possibility. As value distributions vary widely across lists, data-adaptive thresholds that are specifically tuned to the individual lists seem to be a promising approach. This idea was already considered in [YLW+05], but deemed computationally intractable and not pursued much further.

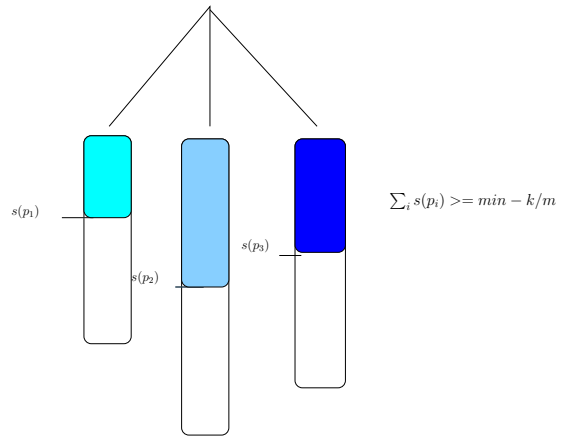

Figure 6.2: Adaptive Thresholds:

We can formally define this optimization problem as follows. Assume that we scan the m input lists to depths d_1, d_2, ..., d_m, and the values at these list positions are $v(d_1)$, $v(d_2)$, ..., $v(d_m)$, respectively. The cost of reading d_i entries from list L_i is $c_i(d_i)$. We need to ensure that we scan deep enough so as not to miss any potential top-k candidate; this mandates the constraint

$$\sum_{i=1}^{m} v(d_i) \leq min\text{-}k$$

with uniform thresholding being a special case (cf. Figure 6.2).

We aim to minimize the total cost of shipping list entries, which is equivalent to minimizing

$$\sum_{i=1}^{m} c_i(d_i)$$

subject to the introduced constraint. For given scan depths d_i we can estimate the resulting $v(d_i)$ by using our probabilistic predictors developed in Chapter 5. It is not too difficult to prove that this problem is NP-hard, as we can reduce the Knapsack problem to our problem.

6.1.1 NP-hardness of the Adaptive-threshold Optimization Problem

The KNAPSACK decision problem can be formulated as follows. Given m items X_i ($i = 1..m$), each with weight w_i and utility u_i, and a weight capacity C, decide for a given constant U if there is a subset $S \subseteq [1..m]$ such that the total utility is at least U, $\sum_{j \in S} u_j \geq U$, and the capacity constraint $\sum_{j \in S} w_j \leq C$ is satisfied.

Given an instance of KNAPSACK, we construct the following instance of the threshold-adaption problem as follows. We consider m lists where the ith

list l_i consists of a single entry with score u_i. The cost to read an entry from list i is w_i. This trivial transformation can obviously be done in polynomial time. Choosing an item X_i in the traditional KNAPSACK terminology corresponds to reading an entry of list l_i.

We claim that (A) a packing for this instance of KNAPSACK has capacity $\leq C$ and utility $\geq U$ if and only if (B) the instance of the threshold-adaption problem has a total cost $\leq C$ and a score $\geq U$.

Proof of (A) $\Rightarrow$ (B): Given a packing of the KNAPSACK instance with capacity $\leq C$ and utility $\geq U$, i.e. we have $X_{i_1}, .., X_{i_k}$, i.e. $l_{i_1}, .., l_{i_k}$, with $w_{i_1} + w_{i_2} + .. + w_{i_k} \leq C$ and $u_{i_1} + u_{i_2} + .. + u_{i_k} \geq U$. Reading the entries from lists $l_{i_1}, .., l_{i_k}$ gives us items with scores $u_{i_1}, ..., u_{i_k}$. Thus, this is a solution to the threshold-adaption problem since we meet the cost bound C and the utility U.

Proof of (B) $\Rightarrow$ (A): Given a solution to the threshold-adaption problem. Let $i_1, .., i_k$ be the lists from which we retrieve an entry. We know that $w_1 + w_2 + .. + w_k \leq C$ and $u_1 + u_2 + .. + u_k \geq U$. Reading from list l_{i_j} is obviously equivalent to choosing item X_{i_j} due to our problem construction. Hence, $\{X_{i_1}, .., X_{i_k}\}$ is a solution to the KNAPSACK problem.

□

Note that this proof only holds under the assumption that we allow data sources to have different access costs. However, this is a reasonable assumption. And although we consider later in our experiments the same network access cost for all peers, the aforementioned assumption is still important since local data access is much cheaper than the access to remote peers. And this case naturally occurs when considering hierarchical query plans.

6.1.2 Heuristic Solution

As we address applications with large m an exact solution is out of the question. However, we can devise practically good approximations based on the following heuristics.

The key idea is to optimize not the sum of the scan depths, but rather the maximum scan depth over the m lists. In a lightly loaded network with all m scans proceeding in parallel on different peers, this objective function would be appropriate for minimizing the latency of this phase. For our actual objective function, minimizing the total bandwidth consumption, it is merely a heuristics, but turns out to be a fairly good approximation in practical settings. If we minimize the deepest scan, i.e., $max_{i=1}^{m} d_i$, we can set all d_i to the same maximum, so that we effectively deal with only one free variable as $d_1 = d_2 = ... = d_m$. We still need to ensure that this choice of d_i's satisfies the constraint. But now we can easily perform a binary search over the possible choices, to find the lowest d_i without violating the constraint. Note that this approach of uniform scan depths usually results in non-uniform local thresholds at which the scans on the individual lists stop. Further note that each step of the binary

search requires evaluating our single-list cost prediction model for each peer. Here we use the model based on linear splines (rather than Poisson mixes) as it is crucial to capture the specific distributions of individual lists and do so with high accuracy. In our implementation, the overhead of these computations is negligible.

As shown in Algorithms 6.1 and 6.2, the idea is to perform two nested binary searches, the outer one to find the cost bounds and the inner one (in the function *findCostThreshold*) to find the thresholds for each peer. The overall runtime is therefore very low, and depending on the network situation it either finds the optimal thresholds or at least good thresholds. Our experiments in Section 6.5 show that the resulting thresholds are far superior to choosing uniform thresholds.

Algorithm 6.1 balanceThresholds(H,$min\text{-}k$)

```
1: input: a set of histograms H, a value threshold min-k
2: output: a set of thresholds for H
3: uniformCosts= 0
4: for each h ∈ H do
5:    c =costs of reading above min-k/|H| (estimated using h)
6:    if c > uniformCosts then
7:       uniformCosts=c
8:    end if
9: end for
10: l = 0, r =uniformCosts
11: while l < r do
12:    m = (l + r)/2
13:    t = 0
14:    for each h ∈ H do
15:       t = t+findCostThreshold(h,m)
16:       if t ≤ min-k then
17:          r = m
18:       else
19:          l = m
20:       end if
21:    end for
22: end while
23: T = ∅
24: for each h ∈ H do
25:    T = T ∪ {(h,findCostThreshold(h,r))}
26: end for
27: return T
```

Algorithm 6.2 findCostThreshold(h,b)

```
1: input: value histogram h, a cost bound b
2: output: the minimal value threshold that meets the cost bound
3: l = r, r =maximum value in h
4: while l + ε < r do
5:    m = (l + r)/2
6:    c =costs of reading above m (estimated using h)
7:    if c < b then
8:       r = m
9:    else
10:      l = m
11:   end if
12: end while
13: return r
```

6.2 Hierarchical Grouping

KLEE and related algorithms employ a flat execution strategy similar to Figure 6.1 (a): all queried peers send their data items directly to the query initiator. This execution model is wasteful for a number of reasons. First, it incurs unnecessary communication. Consider, for example, a query with one very large and several small input lists residing on different peers. It would be better to perform the top-k query at the peer with the large list, have the small peers ship their items to the large peer, and only send the final result to the query initiator. Second, the peers compete for network bandwidth, as all of them send their data items to the query initiator forming the top-k aggregation. If instead several peers aggregate data from other peers and only send their aggregated results to the querying peer, the total bandwidth consumption can be reduced.

We apply a hierarchical grouping of peers in the second phase of the algorithm to reduce transfer costs [NM07]. Figure 6.1 (b) illustrates an example execution plan for a query with $m = 4$ input lists L_1 through L_4 on four different peers p_1 through p_4. Instead of querying all four peers for their list items with value above the threshold min-$k/4$, the query initiator contacts only peer p_1, which itself contacts p_2 and p_3 with a threshold of min-$k/3$ (the last third of the threshold remains at p_1). The peer p_3 subsequently forwards the request to its children in the execution plan, again dividing the threshold by the respective number of children. For peer p_3, the new threshold is $min\text{-}k/(3 * 2)$, as p_3 has two children, including list L_3 which is stored locally. Note that the threshold for the relatively large peer p_2 is higher than the threshold in a flat execution, $min\text{-}k/4$, reducing the number of items sent. When p_3 has received all items from its children, it aggregates them with the items of its own list and sends the result to its parent in the execution plan. As some items may occur in multiple lists, the number of items sent to p_1 is typically less than in a flat execution.

Using such a hierarchical grouping can improve the query execution, but, depending on the sizes and value distributions of the input lists, may also ad-

versely affect performance by adding latency and transfer cost (as data must pass through more than one peer). Therefore the hierarchical grouping must be constructed by a query optimizer that computes the cost of the candidate trees and chooses the best alternative. In the following, we first discuss a dynamic programming method for finding the best hierarchical structure and then discuss a fast heuristics to handle larger problems.

6.2.1 Dynamic Programming Approach

One way to find the optimal hierarchical structure is to employ dynamic programming (DP) [CLRS01]. Note that we only optimize the second phase of the algorithms; so the *min-k* threshold is already known, we only have to organize the aggregation of data items. The cost of each aggregation step is determined by the costs its slowest input (*max*) and the bandwidth limitations for getting the input data to aggregating peer (basically a weighted *sum*).

Figure 6.3 shows the optimization algorithm in pseudocode; the algorithm applies dynamic programming in a top-down formulation with memoization. The dynamic programming table maps $(lists, min\text{-}k) \rightarrow (peer \rightarrow plan)$, i.e., for each combination of input lists and *min-k* threshold, we compute and keep the optimal plan for each possible target peer where the subquery result could reside.

In our distributed setting, the placement of data also has to be taken into account. This leads to the following optimization process:

1. The algorithm always considers all possible peers as location for the result, i.e., it operates on sets of plans – one plan for each possible peer where the final result could reside.

2. A (sub-)problem can always be solved by using a flat execution, i.e., aggregating the input peers at the target (lines 7-8).

3. If the problem consists of more than one input peer, the aggregation can instead be performed hierarchically: the problem is split into smaller problems whose results are then combined (lines 10-20).

4. As it might be better to perform the entire aggregation at one peer and merely ship the results, the algorithm considers the cost of this case (lines 21-25).

To assess the quality of an execution tree, the algorithm estimates its transfer cost. For the transfer cost, the number of items transferred from a group of peers to their parent is estimated using the statistical prediction model of Chapter 5. As we are optimizing a plan for a specific set of peers, we are using the more accurate linear-spline model for each list, and additionally the estimated cardinality of each list. For larger m when the optimization procedure itself becomes more expensive, we can resort to the faster Poisson-mix model; alternatively, we can use a faster heuristics (see below) and/or use sampling (see Section 6.3). It is difficult to determine tight bounds for the algorithm

complexity, as search space pruning depends on the concrete problem. Note that the pseudo code is simplified, it shows the search space organization but hides several implementation details. The DP algorithm can be implemented with an upper bound of $O(m4^m)$. Unfortunately $\Omega(2^m)$ is a lower bound which makes using DP infeasible for large m.

Algorithm 6.3 buildHierarchy(I,*min-k*)

```
1:  Input: a set of data-item lists I; value threshold min-k
2:  Output: a set of optimal execution plans, one for each peer
3:  if (I, min-k) has already been solved then
4:    return known solution
5:  end if
6:  b = empty plan set
7:  for each p ∈ peers do
8:    b[p] = flat aggregation of I at p, threshold min-k
9:  end for
10: if |I| > 1 then
11:   for eachP = {I_i ⊂ I}, P partitioning of I do
12:     I' = {buildHierarchy(I_i, min-k/|P|)|I_i ∈ P}
13:     for each p ∈ peers do
14:       I_p = {i[p]|i ∈ I'}
15:       a =aggregation of I_p at p
16:       if a.costs< b[p].costs then
17:         b[p] = a
18:       end if
19:     end for
20:   end for
21:   for each p_1, p_2 ∈ peers do
22:     if transfer(b[p_1], p_2).costs< b[p_2].costs then
23:       b[p_2] =transfer(b[p_1], p_2)
24:     end if
25:   end for
26: end if
27: store b as solution for (I, min-k) in DP table
28: return b
```

6.2.2 Fast Heuristics

The DP method finds the optimal hierarchical structure, but its run-time may be prohibitively high.

DP (in particular the partitioning step (line 11), trying out all partitionings) becomes infeasible when the number of lists to aggregate is too large. To avoid the exhaustive search, we use a fast heuristics to find a good partitioning. The hierarchical structure is basically a divide-and-conquer strategy for the aggregation; therefore, we want to partition the lists such that the resulting

partitions exhibit approximately equal costs. In our cost model, lists with similar cardinality will cause similar effort; so we heuristically partition the lists I as follows:

$$\begin{aligned} S_I &= I \text{ sorted by cardinality above } \textit{min-k}/|I| \\ O_I &= \text{every odd entry of } S_I \text{ (with asc. cardinality)} \\ E_I &= \text{every even entry of } S_I \text{ (with desc. cardinality)} \end{aligned}$$

We expect that O_I and E_I are similar, e.g. O_I and E_I would already be a good partitioning. However the cardinalities can vary widely; therefore we consider moving some of the smaller lists (tail of O_I, head of E_I) from one partition to another. We concatenate O_I and E_I (which is sorted by reverse cardinality), and cut the resulting list at any position to get partitioning candidates. The resulting search space is no longer exponential, allowing for an implementation in $O(m^2)$ using search space pruning. This heuristics works very well in practice and allows very fast construction of competitive execution trees even for large numbers of input lists.

6.3 Site Sampling

For distributed queries that span hundreds of peers, none of these techniques is sufficient to ensure interactive performance and we rather consider a sampling approach that operates only on a small fraction of randomly chosen input lists.

Recall our estimation model from Chapter 5, based on a score distribution model for each list, the convolution over m lists, and a first-order approximation of the expectation of the order-k statistics for the value $\textit{min-k}$ when aggregating n items over all m lists. Now we introduce two kinds of sampling:

1. Instead of considering all n items per list, we consider only the top n' items in a list.

2. Instead of considering all m lists, we consider only a sample of m' lists.

For method 2 the sample may be chosen uniformly or in a biased manner. In the latter case, we study the selection of the m' lists with the highest value sums $w_i := \sum_{j=1}^{n} val_i(d_j)$. We assume that we know the fraction of the total value sum that these m' lists accumulate:

$$\varphi := \sum_{i=1}^{m'} w_i / (\sum_{i=1}^{m} w_i)$$

(without loss of generality, assume that the m' lists are numbered $1...m'$).

We can now estimate the sampling-based min-k value when considering only the top n' items in m' lists. The linear error $|\textit{min-k}(m,n) - \textit{min-k}(m',n')|$ is a measure of the accuracy of the sampling-based top-k algorithm, and we use this error measure for calibrating the choices of m' and n'.

In the analysis we make a number of model simplifications for tractability:

1. We assume that the X_i are identically and independently distributed.
2. At several points we consider only expectations rather than full distributions.
3. We model the effect of sampling in a conservative manner, i.e., overestimating its error.

We sample m' lists and n' items from each list (which are typically non-disjoint across lists). We estimate the expected number of lists, m'', in which we see an item, and the expected number of distinct items, n'', seen in all lists. We make the conservative error of assuming that the n' items are uniformly drawn among the items in a list (whereas in reality we draw the top-n' items).

With uniformly chosen m' lists: P[item seen in q out of m' lists] $= p_{seen}(q) = \binom{m'}{q}(\frac{n'}{n})^q(\frac{n-n'}{n})^{m'-q}$ with expectation $E_{seen} = m'n'/n' := m''$. The probability that we see item d in at least one list is $1-(1-\frac{n'}{n})^{m'}$ and the expected number of distinct items seen in all lists together s $E_{dist} = (1-(1-\frac{n'}{n})^{m'})n$. Now we assume that each of the n (or E_{dist}) items is seen in exactly m'' lists and estimate $min\text{-}k(m'', n')$ using the available cost model of 5.

With the non-uniform sampling strategy that selects the m' "heaviest" lists, the analysis of E_{seen} and E_{dist} remains the same, but we adjust the Poisson-mixture parameters in the estimation of $min\text{-}k(m'', n')$ as follows. We assume that all m' lists have the same value distributions but together constitute fraction φ of the overall value sum over all lists. Thus, the expected value of an item in one of the m' lists is larger than the expected value in a model with all lists having equal weight by the factor $\varphi m/m' =: \rho$, the "boost factor". We then adjust the parameters of the per-list Poisson-mixture model to have this boosted expectation. The expectation of the non-weighted Poisson mix is $\alpha\beta + (1-\alpha)\gamma$. This easiest way of boosting the expectation then is by setting $\beta' := \rho\beta$ and $\gamma' := \rho\gamma$, yielding the expectation $\rho(\alpha\beta + (1-\alpha)\gamma)$. After this adjustment, we can use our top-k cost model for estimating the adjusted $min\text{-}k(m'', n')$.

6.4 Dealing with Network Failures

Our query execution strategies assume that the network is stable for the duration of the query in order to have a clear semantics for the query result. Peer failures and other aspects of network dynamics (e.g., traffic bursts that slow down peers or the churn phenomenon in P2P systems) pose extra difficulties. While a comprehensive discussion of these issues is beyond the scope of this work, we offer some simple steps to increase the robustness of our methods, based on standard techniques for monitoring the liveness of peers (e.g., "heart-beat" messages and timeouts).

When the query originator fails, the query can be aborted anyway; if the failure is transient, the query originator can resubmit the query after its restart. When a node fails that was involved in message routing (e.g., an intermediate node in DHT-based routing) but is not involved in the query execution, we

employ whatever routing redundancy the underlying network provides (there is ample literature on dynamic re-routing in the networking and P2P systems community).

The remaining, not so straightforward, case is when one of the peers fails that is involved in the query execution tree. When a peer realizes that its parent has failed, the techniques of [APV06] can be applied (cf. Figure 6.3): the orphaned peer sends its results either directly to the query initiator or to some known ancestor in its caller tree. Conversely, when a peer realizes that one of its children has failed, it may either find alternative routes to reach its affected grandchildren or it could view the entire subtree as unavailable. Such steps may even include dynamic re-optimizations (cf. Figure 6.4), e.g., to adjust thresholds. Exploring approaches along these lines is left for future work.

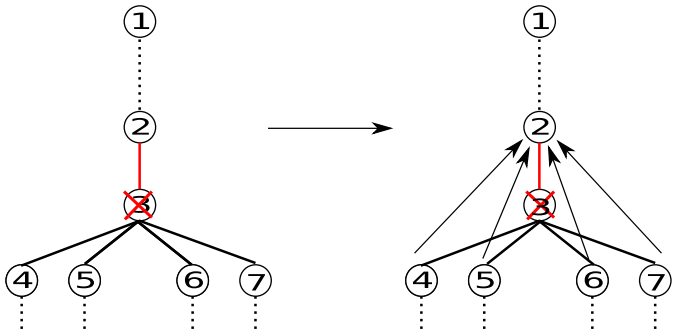

Figure 6.3: Children nodes send data to grand-parent directly.

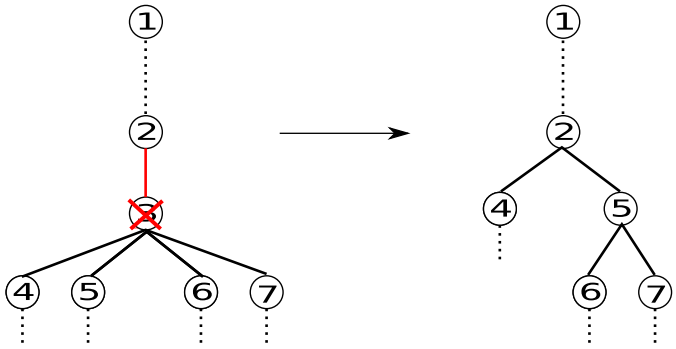

Figure 6.4: Dynamic re-organization in case of node failures.

6.5 Experiments

6.5.1 Setup

We have implemented all algorithms and our testbed in C++. To obtain reproducible and comparable results, we simulate the network part. Running

experiments over multiple peers in a real wide-area network would suffer from unpredictable and irreproducible interference by other applications. For network latency, we use the following parameters. We assume a packet size of 1 KByte with a round-trip time (RTT) of 150ms to model the latencies for data transfer sizes up to 1 KByte. For larger transfers which are dominated by bandwidth rather than RTT, we assume the latencies to fall out from a bulk data transfer rate of 800 KBits/s. These numbers represent the averages reported in [SL00] for sending large files between the Stanford Linear Accelerator Center (SLAC) and nodes in Lyon, France, using NLANR's iPerf tool [Tir03].

The query response time combines CPU times, disk IO times, and network communication times. A peer that has m input lists observes a total query response time of

$$RT = \frac{\sum_{i=1}^{m} \#bytes(P_i)}{available\ bandwidth} + max_i(\text{latency of } P_i)$$

seconds, where $\#bytes(P_i)$ is the total transfer volume of peer P_i and the overall available bandwidth is shared among all peers (thus disregarding specifics of the network topology, but giving each peer only one out of m shares is conservative).

Algorithms under Comparison

TPUT is the three-phase uniform threshold algorithm [CW04]. We do not consider the variant of TPUT that uses a compression technique based on hash array encoding to decrease the network bandwidth consumption. We consider it an orthogonal issue to apply compression techniques to any of the investigated algorithms.

KLEE (cf. Chapter 4) is an extension of TPUT that employs histograms and Bloom filters to increase the $min\text{-}k/m$ threshold. It is an approximate algorithm, i.e., it does not guarantee to find the exact top-k query results. The overall performance of KLEE depends on a parameter c that determines the number of Bloom filters that are transferred in the first round as a fraction of the total value mass of an input list. We set $c = 5\%$. We use only the three-phase KLEE variant, and disregard the KLEE-4 variant as its additional filtering step would be orthogonal to the issues studied here.

GRASS-1 is an extension of KLEE that uses our adaptive thresholding described in Section 6.1, i.e., phase 2 uses non-uniform value thresholds.

GRASS-2 uses hierarchical query execution plans as computed by the optimization techniques of Section 6.2, in addition to the adaptive-threshold technique of GRASS-1. In the experiments, we use the dynamic-programming algorithm for m up to 10 and switch to the fast heuristics for higher m.

GRASS-3 additionally utilizes the sampling techniques described in Section 6.3. We sample a number of peers, in descending order of value mass, so that

our *min-k* estimate predicts a maximum error of at most 20% compared to the *min-k* estimate if we ran the query on all peers. In the experiments this resulted in typically selecting between 10 and 30 percent of the peers involved in a query.

Approximate vs. Exact Mode

KLEE has explicitly been designed as an approximate algorithm. However, KLEE and all non-sampling methods can be turned into exact algorithms by adding a random-lookup phase at the end of each algorithm. The resulting algorithms can be considered as TPUT variants flavored with KLEE's techniques plus our optimization techniques. Our experiments showcase the effectiveness and efficiency for both operation modes of the algorithms under test.

Datasets

The **WorldCup** HTTP server log collection[2] consists of about 1.3 billion HTTP requests recorded during the 1998 FIFA soccer world cup. We distributed the data across peers by time intervals, each day is assigned to a different peer. The task is to identify the top-k most popular files within certain time intervals, for example, a period of one month resulting in a query over 30 peers.

AOL Query Log[3]: This search-engine query log consists of ∼20M queries collected from ∼650k users over three months. We have considered all `(userid, terms, date)` triplets where the userids provide a stable mapping from queries to users over the entire time period. We have grouped the queries by userid and, for each user, created all possible term pairs from her queries after applying stemming and stopword elimination. We finally created 5000 peers from the users with the highest numbers of different term pairs. Here, a top-k query consists of a set of m users and the task to find the top-k most frequent term pairs that occur in the queries issued by the users over the complete time interval.

The **Retail Benchmark** consists of retail market basket data from an anonymous Belgian retail store [BSVW99]. A set of 100 peers was generated by randomly assigning each of the $\sim 88k$ transactions to exactly one peer, modeling a situation as if the transactions had occurred at distributed shopping sites. At each peer, we generated all possible triplets of basket items present in any of the transactions, yielding a total number of $51,788,094$ ($16,769,821$ distinct) triplets. As for queries, we are interested in finding the globally most frequent triplets, using only a subset of the 100 peers (i.e., retail stores).

Performance Metrics

Cost factor bandwidth: Total number of bytes transferred between the query initiator and the peers that are involved in executing a query.

[2] *http://ita.ee.lbl.gov/html/contrib/WorldCup.html*
[3] *http://www.gregsadetsky.com/aol-data/*

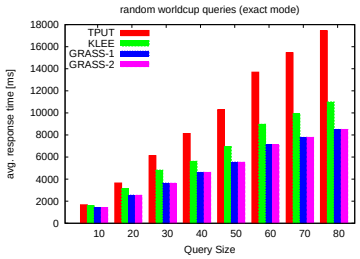

Figure 6.5: Worldcup results in exact mode

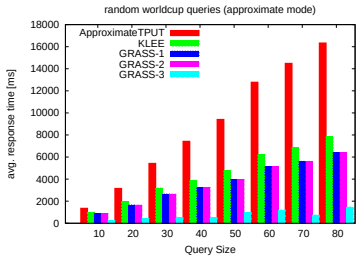

Figure 6.6: Worldcup results in approximate mode

Cost factor query response time: Estimated elapsed "wall-clock" time for the benchmarks, using measured numbers for the CPU time, disk IO time, and network traffic, and using our network simulation model (see above) for deriving elapsed time.

Quality factor relative recall: Overlap between the top-k results produced in the experiments by approximate algorithms and the true global top-k results produced by an exact method. By the exact nature of the algorithms, both the original TPUT and the exact KLEE variants have a relative recall of 1.

6.5.2 Results

Figure 6.5 illustrates the average query response times for the WorldCup benchmark with all algorithms operating in exact mode; more complete results are given in Table 6.1. Each point in the chart is computed by averaging over 10 independently chosen random queries for the given number of peers (i.e., appropriately chosen query parameters). GRASS-1 and GRASS-2 reduce the average response time by more than a factor of two compared to TPUT. KLEE outperforms TPUT, but in turn is outperformed by GRASS-1 and GRASS-2.

Figure 6.6 and Table 6.2 show the query response times with the algorithms operating in approximate mode. The improvements by GRASS-1 and GRASS-2 are remarkable, compared to ApproximateTPUT. GRASS-1 performs more or less identical to GRASS-2 in both operation modes, the reason being that in this setting all input lists were fairly short and had very high positive correlation so that the hierarchical grouping could not really improve over flat execution plans. In approximate operation mode, our sampling algorithm GRASS-3 clearly outperforms all other algorithms. It dramatically reduces the query response times while maintaining acceptable relative recall (see Table 6.2).

Figure 6.7 (cf. Table 6.3) shows the average query response times for the AOL benchmark where all algorithms operate in exact mode. Throughout the experiment GRASS-2 shows the best performance. KLEE and GRASS-1 per-

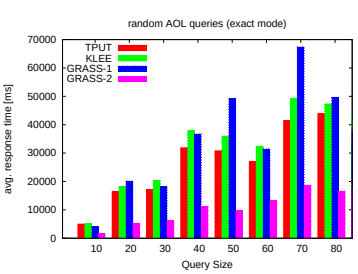

Figure 6.7: AOL results in exact mode

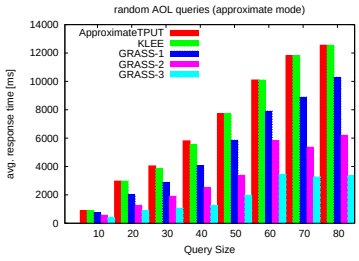

Figure 6.8: AOL results in approximate mode

form worse than TPUT. The potential benefits of threshold adaption are outweighed by the increased costs for additional random lookups to eventually determine the exact results. In this setting with much more heterogeneous and thus more interesting value distributions at different peers, GRASS-2 greatly benefits from its hierarchical grouping capabilities, and the chosen execution trees outperformed GRASS-1 by a large margin.

When the algorithms operate in approximate mode (Figure 6.8), the results show a draw between ApproximateTPUT and KLEE. Our algorithms (GRASS-1, GRASS-2, and GRASS-3), in contrast, reduce the response time by a large factor. GRASS-3 performs particularly well, while maintaining a reasonable relative recall (cf. Table 6.4). By nature of the benchmark, this setting allows us to consider very big queries of size (i.e., number of involved peers) 200,300, and 400. GRASS-3 is able to reduce the query response time of the baseline methods TPUT and KLEE by a factor of about 7 for the query over 400 input lists and even by a factor of more than 40 for the query over 300 lists while maintaining a relative recall of 66% and 40%, respectively.

Figure 6.9 shows the average query response times for the Retail benchmark when all algorithms run in exact mode. Similarly to the results for the AOL benchmark, TPUT performs well compared to KLEE and GRASS-1, which, again, suffer from the additional random lookup phase, as they were not designed for an exact mode of operation. GRASS-2, on the other hand, again shows excellent performance and is the clear winner on this benchmark, too.

Figure 6.10 reports the average query response times (see Table 6.6 for the complete results) when all algorithms operate in approximate mode; Table 6.6 summarizes the average response times, the average bandwidth consumptions, and the average recall. The KLEE-based variants are superior to TPUT for this scenario. Adaptive thresholding, hierarchical query execution plans, and sampling further improve KLEE's performance.

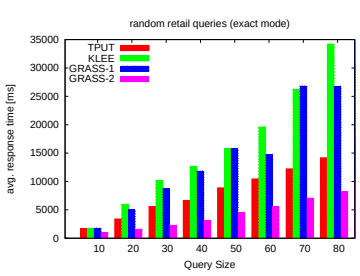

Figure 6.9: Retail results in exact mode

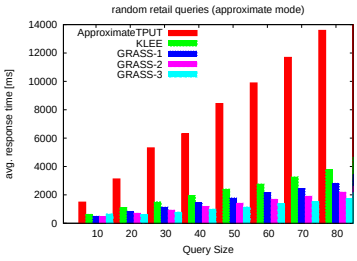

Figure 6.10: Retail results in approximate mode

	TPUT		KLEE		GRASS-1		GRASS-2	
size	**time [s]**	**bytes [kB]**	**time [s]**	**bytes [kB]**	**time [s]**	**bytes [kB]**	**time [s]**	**bytes [kB]**
20	3.65	340.19	3.14	293.31	2.55	234.19	2.55	234.19
40	8.13	788.45	5.62	544.73	4.61	443.93	4.61	443.93
60	13.69	1,344.90	8.97	884.11	7.11	697.80	7.11	697.80
80	17.45	1,720.35	10.99	1,089.92	8.50	840.66	8.50	840.66

Table 6.1: Worldcup results in exact mode

6.5.3 Discussion

Overall, GRASS-2 is the best performing method. It is superior to all other competitors in most cases, and still very competitive in the few situations in which it is outperformed by one of the other methods. GRASS-2 combines the benefits of the adaptive thresholding and the hierarchical grouping techniques, leveraging our novel cost prediction model for both. Compared to TPUT and KLEE, GRASS-2 typically gains a factor of 2, sometimes even up to a factor of 10 for individual queries (not shown in the averaged results in the figures and tables). Compared to GRASS-1, the full-fledged GRASS-2 algorithm typically wins by 20 to 50 percent, still a significant margin. However, GRASS-1 is much less robust than GRASS-2: there are cases when GRASS-2 is 3 or more times faster than GRASS-1, and GRASS-1 even loses to TPUT or KLEE in some cases. This shows that the adaptive thresholding alone is too brittle as a heuristics, and the cost-based optimization of GRASS-2 is not only worthwhile but crucial for consistently good performance.

GRASS-3 has even shorter response times than GRASS-2, but the two methods are actually incomparable as GRASS-3 is inherently approximate and typically exhibits a non-negligible loss in relative precision. Notwithstanding this observation, GRASS-3 is the method of choice for very high m. And it achieves a very impressive quality/cost ratio. For queries with 400 peers, GRASS-3 outperforms all other methods, including GRASS-2, by a factor of 5 while still retaining a decent result quality with a relative recall above 60 percent.

Although the issue of exact vs. approximate results is orthogonal to our algorithmic contributions of this work, we think it is worthwhile pointing out that the approximate variants of GRASS-2 is a particularly intriguing algorithm

	ApproximateTPUT			KLEE		
size	time [s]	bytes [KB]	recall	time [s]	bytes [KB]	recall
20	3.16	295.83	0.96	1.95	178.78	0.94
40	7.43	723.17	1.0	3.87	374.35	0.96
60	12.78	1,258.27	0.98	6.22	612.58	0.96
80	16.34	1,613.63	0.98	7.84	778.06	0.96

	GRASS-1			GRASS-2			GRASS-3		
size	time [s]	bytes [KB]	recall	time [s]	bytes [KB]	recall	time [s]	bytes [KB]	recall
20	1.66	149.61	0.93	1.66	149.61	0.93	0.43	23.59	0.61
40	3.22	309.35	0.95	3.22	309.35	0.95	0.50	31.14	0.59
60	5.12	503.07	0.96	5.12	503.07	0.96	1.20	102.16	0.63
80	6.40	634.66	0.95	6.40	634.66	0.95	1.46	129.87	0.61

Table 6.2: Worldcup results in approximate mode.

	TPUT		KLEE		GRASS-1		GRASS-2	
size	time [s]	bytes [kB]	time [s]	bytes [kB]	time [s]	bytes [kB]	time [s]	bytes [kB]
20	16.35	1,614.66	18.14	1,796.28	20.12	1,995.15	5.29	948.01
40	31.84	3,166.01	37.98	3,783.55	36.75	3,668.09	11.10	2,006.68
60	27.06	2,687.88	32.39	3,228.38	31.49	3,142.20	13.17	2,126.07
80	43.88	4,370.91	47.33	4,724.74	49.61	4,961.08	16.63	3,017.14
100	39.04	3,886.11	41.61	4,152.94	37.13	3,709.20	16.41	2,984.45
200	60.74	6,053.75	68.46	6,846.06	53.99	5,399.93	36.13	4,882.66
300	93.18	9,295.25	96.90	9,699.27	82.54	8,265.04	58.13	7,590.90
400	127.55	12,732.50	128.51	12,867.40	101.73	10,193.92	78.62	9,608.18

Table 6.3: AOL results in exact mode

for many practical applications. It is typically a factor of 2 or 3 faster than its exact counterpart (sometimes even by a higher factor as m increases), but consistently achieves a relative recall above 90 percent or higher – an excellent result quality that would be perfectly acceptable by most applications of top-k querying.

	ApproximateTPUT			KLEE		
size	time [s]	bytes [KB]	recall	time [s]	bytes [KB]	recall
20	2.97	276.70	1.00	2.97	278.65	1.00
40	5.81	560.95	1.0	5.56	540.21	0.98
60	10.09	989.50	0.99	10.09	995.37	0.99
80	12.55	1,234.87	0.98	12.55	1,242.62	0.98
100	19.14	1,893.55	0.98	19.14	1,903.31	0.98
200	45.68	4,547.76	0.99	45.68	4,567.34	0.99
300	86.01	8,580.83	1.00	73.66	7,375.43	0.99
400	119.96	11,976.24	1.00	119.96	12,015.29	1.00

	GRASS-1			GRASS-2			GRASS-3		
size	time [s]	bytes [KB]	recall	time [s]	bytes [KB]	recall	time [s]	bytes [KB]	recall
20	2.04	185.58	0.99	1.27	159.24	0.99	0.88	95.18	0.95
40	4.05	389.24	0.97	2.53	338.41	0.96	1.25	159.27	0.89
60	7.91	776.89	0.97	5.82	701.82	0.97	3.45	404.34	0.90
80	10.29	1,016.59	0.97	6.22	961.96	0.97	3.35	531.29	0.88
100	13.72	1,361.46	0.97	7.44	1,267.98	0.97	4.41	654.03	0.88
200	29.57	2,956.98	0.98	26.91	2,796.11	0.98	3.00	214.05	0.57
300	48.64	4,873.30	0.97	43.53	4,699.13	0.98	1.65	117.92	0.40
400	68.43	6,862.03	0.98	63.18	6,602.09	0.98	17.78	1,753.67	0.66

Table 6.4: AOL results in approximate mode.

	TPUT		KLEE		GRASS-1		GRASS-2	
size	time [s]	bytes [kB]	time [s]	bytes [kB]	time [s]	bytes [kB]	time [s]	bytes [kB]
20	3.38	313.54	5.97	870.22	5.07	779.99	1.61	507.48
40	6.66	641.44	12.65	1,836.55	11.85	1,757.10	3.10	1,072.00
60	10.44	1,019.45	19.62	2,837.32	14.72	2,345.32	5.53	1,817.87
80	14.16	1,391.56	34.22	4,593.06	26.76	3,850.58	8.17	2,608.02
100	17.70	1,744.96	35.47	5,010.64	35.58	5,026.48	10.46	3,327.03

Table 6.5: Retail results in exact mode

	ApproximateTPUT			KLEE		
size	time [s]	bytes [KB]	recall	time [s]	bytes [KB]	recall
20	3.12	292.42	0.97	1.10	386.15	0.93
40	6.31	611.07	1.0	1.95	768.82	0.92
60	9.89	969.18	0.98	2.77	1,149.08	0.92
80	13.60	1,339.76	0.99	3.77	1,546.71	0.91
100	16.86	1,666.22	0.98	4.65	1,923.35	0.92

	GRASS-1			GRASS-2			GRASS-3		
size	time [s]	bytes [KB]	recall	time [s]	bytes [KB]	recall	time [s]	bytes [KB]	recall
20	0.83	359.62	0.91	0.71	350.68	0.903	0.62	302.98	0.85
40	1.45	718.33	0.91	1.19	708.85	0.904	0.99	600.23	0.86
60	2.15	1,087.42	0.90	1.69	1,076.32	0.901	1.38	907.77	0.84
80	2.82	1,451.17	0.90	2.20	1,439.62	0.891	1.74	1,202.05	0.84
100	3.45	1,802.72	0.90	2.64	1,790.07	0.89	2.19	1,539.80	0.85

Table 6.6: Retail results in approximate mode.

Chapter 7

Probabilistic Guarantees

KLEE and GRASS are approximate algorithm that can be turned into exact mode by using random lookups at the end of the second phase, i.e. using TPUT's third phase. Our performance evaluation shows that major performance gains can be achieved when running in approximate mode while the relative recall (and thus also precision) is still at a very high level.

However, these experimental results heavily depend on the peculiarities of the underlying data. To overcome these limitations of a meaningful analysis, this chapter presents probabilistic guarantees for the result quality of the approximate versions of TPUT, KLEE, and GRASS.

Recall that TPUT [CW04] uses the following three phase structure:

1: The query initiator, P_{init}, retrieves the top k items from each list. P_{init} aggregates the scores and ranks the documents according to their partial scores (worstscores, i.e. aggregation of scores seen so far). Let *min-k* denote the partial score of the document that is currently at rank k.

2: P_{init} sends the threshold $min\text{-}k/m$ to the involved peers. Each peer returns all items with score above $min\text{-}k/m$.

3: P_{init} retrieves all missing scores for the candidate items.

TPUT is exact, i.e. it calculates the true top-k result. KLEE [MTW05a] is an approximate version of TPUT that does not employ the third phase and, in addition, uses compression techniques based on Bloom filters and a technique to filter out unpromising candidates.

For now we assume that the compression techniques applied in KLEE are perfect, i.e. do not cause any false positives. It is straightforward to configure Bloom filters to provide this error-free behavior with high probability [Blo70]. So, we actually consider the approximate version of TPUT (without the third phase) and disregard KLEE's optimization techniques.

7.1 Problem Statement

We consider the state of the algorithm after the second phase.

Let $high_i$ denote the score at the current scan depth pos_i of index list L_i. The bestscore is the upper bound of an items's score. Let $E(d)$ denotes the set of index lists in which we have seen d so far. Then, $bestscore(d) = worstscore(d) + \sum\{s_i | i \notin E(d)\}$. For now we will consider only summation as the aggregation function.

All items with bestscore $< min\text{-}k$ have been pruned away. This is the standard pruning technique that does not introduce errors.

The error in the relative recall at this stage of the algorithm is caused by the ordering of the items since this ordering is based on incomplete information (i.e. not fully evaluated candidate items). A typical situation after phase 2 is depicted in Figure 7.1.

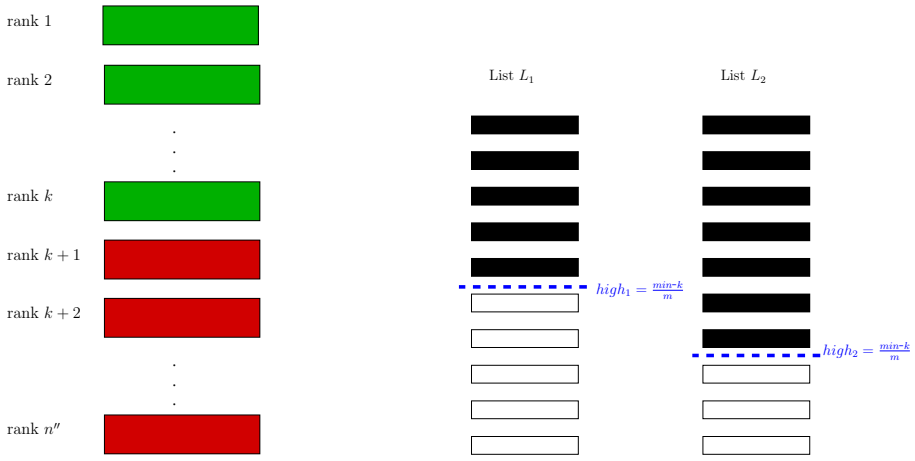

Figure 7.1: (Left): The first k entries of the current ranking are considered to be the current top-k estimate. The items with rank $> k$ are the candidates. We assume the item at rank $k+1$ to have the highest chance to get into the final top-k. (Right): The items in the (unseen) tails of all index lists have a score smaller than $min\text{-}k/m$. This is due to the score threshold propagation in the second phase and essential for the reasoning about quality guarantees that we present here.

Assumption: The item that is currently at rank $k+1$ has the highest chance to get into the top-k result (cf. Figure 7.1). In general, an item at rank $> k+1$ can have a higher chance to get into the top-k than the item rank $k+1$ if it has been seen in fewer index lists than the item at rank $k+1$ and thus may achieve a higher final score. In our scenario it does not make sense to distinguish between the different index lists since they have the same $high_i$ values (cf. Figure 7.1) and we do not consider the score distributions in the tails of the index lists. Since we try to derive a general probabilistic guarantee, we cannot treat items individually. In this setting, assuming that the item at rank

$k+1$ has the highest chance to get into the top-k is a meaningful assumption.

7.2 Reasoning about Result Quality

As in Chapter 5, we assume that we read n' items from all m lists. For each document, the expected number of lists in which we have seen it is $mn'/n =: m'$. The expected number of distinct items seen is

$$n'' := 1 - (1 - n'/n)^m n$$

Using order statistics [DN03] we can derive an approximation of the *min-k* value:

$$E[S_{n-k+1}] \approx F_S^{-1}(\frac{n'' - k + 1}{n''})$$

where F^{-1} is the quantile function.

Obviously, after the second phase, $high_i = min\text{-}k/m$ for $i = 1, .., m$. Note that we do not consider the case of non-uniform thresholds here.

Consider a particular document d. We define $\delta(d) := min\text{-}k - worstscore(d)$. Then, the probability that this documents gets into the top-k list is $p(d) = P[\sum\{s_i | i \notin E(d)\} > \delta(d)]$.

Unlike TA-sorted (NRA) with probabilistic guarantees [TWS04], we are not interested in proceeding with the list scans but consider the final ranking (after phase 2).

We try to derive an upper bound for $p(d)$ by considering the probability that the item d at current rank $k+1$ with partial score S_{n-k} gets into the top-k.

Lemma 7.2.1 *Consider a top-k query over m index lists, and the ranking of n'' items that represents the state of the TPUT algorithm after the second phase. Then*

$$p_{upper} := \frac{m - m'}{m} * \frac{1}{1 - \frac{S_{n-k}}{min\text{-}k}} \tag{7.1}$$

is an upper bound of the probability that a true top-k item is at rank $> k$, i.e. currently not in the top-k list.

Proof Consider an item d observed while scanning the index lists. Then

$$p(d) = P[\sum\{s_i | i \notin E(d)\} > \delta(d)] \tag{7.2}$$

$$\leq P[\sum\{s_i | i \notin E(d)\} > (min\text{-}k - S_{n-k})] \tag{7.3}$$

$$\leq_{Markov} \frac{E[\sum\{s_i | i \notin E(d)\}]}{min\text{-}k - S_{n-k}} \tag{7.4}$$

$$\leq \frac{(m - m')\frac{min\text{-}k}{m}}{min\text{-}k - S_{n-k}} \tag{7.5}$$

$$= \frac{m - m'}{m} * \frac{1}{1 - \frac{S_{n-k}}{min\text{-}k}} =: p_{upper} \tag{7.6}$$

From (7.2) to (7.3): replace the general $\delta(d)$ for a particular document by the δ of the document at rank $k+1$.
From (7.3) to (7.4): follows directly from Markov's Inequality.
From (7.4) to (7.5): replace the expected score a document can get in the tails of the index lists by $(m - m') * (min\text{-}k/m)$, where m' is the expected number of lists in which we have seen a particular item.

□

For the relative recall of the top-k results we derive that

$$P[recall = r/k] = \binom{k}{r}(1 - p_{miss})^r p_{miss}^{(k-r)} \tag{7.7}$$

where r denotes the number of correct results in the approximate top-k, and p_{miss} is the probability that we miss a true top-k item.

Theorem 7.2.2 *Consider a top-k query over m index lists. Then, the approximate version of TPUT, and KLEE (with perfect compression, i.e. no false positives) have an expected recall* $\geq 1 - p_{upper}$.

Proof From Lemma 7.2.1 we know that p_{upper} is an upper bound of the probability that a true top-k item is at rank $> k$, i.e. currently not in the top-k list. Using Equation (7.7) we can now derive $E[recall] \geq 1 - p_{upper}$:

$$E[recall] = \sum_{r=0}^{k} P[recall = r/k] * r/k \tag{7.8}$$

$$= \left(\binom{k}{r}(1 - p_{miss})^r p_{miss}^{(k-r)}\right) * r/k \tag{7.9}$$

$$= 1 - p_{miss} \tag{7.10}$$

$$\geq 1 - p_{upper} \tag{7.11}$$

□

Besides the expectation of the recall, one is usually interested in the probability that the recall (or relative recall) is above a particular threshold.

Theorem 7.2.3 *Consider a top-k query over m index lists. Then, the approximate version of TPUT, and KLEE (with perfect compression, i.e. not lossy) achieve a relative recall of at least* r/k *with probability*

$$P[recall > r/k] \geq 1 - I_{p_{upper}}(k - r, r + 1)$$

where $I_{.}(.,.)$ *is the regularized incomplete beta function.*

Proof As the *cdf* of a random variable with binomial distribution is the regularized incomplete beta function $I_.(.,.)$ we know that

$$P[recall > r/k] = 1 - I_{p_{miss}}(k - r, r + 1)$$

From Lemma 7.2.1, we know that $p_{miss} \leq p_{upper}$, thus

$$P[recall > r/k] \geq 1 - I_{p_{upper}}(k - r, r + 1)$$

□

Furthermore, we can derive Chernoff-Hoeffding bounds [Hoe63, Was04] for the distribution of the recall.

Theorem 7.2.4 *For all $\psi > 0$ we have*

$$P[|recall - E[recall]| > \psi] \leq 2e^{-2k\psi^2}$$

Proof The proof follows directly from applying the Chernoff-Hoeffding inequality.

For illustration, Figure 7.2 shows for several values of $(m - m')/m$ and the score differences between the items at rank k and $k + 1$ the expected recall. $(m - m')/m$ denotes the expected percentage of index lists in which we have not observed a particular item. Apparently, with decreasing value of $(m - m')/m$ the expected recall increases since the uncertainty about the current scores decreases. Another observation is that the expected recall increases if the score differences between items at rank k and $k + 1$ increases. This is due to the fact that a larger "gap" between rank k and $k + 1$ decreases the probability that the rank $k + 1$ item will make it into the top-k, i.e. p_{upper} decreases.

7.3 Random Lookups After Probabilistic Pruning

We can employ the probabilistic pruning technique as described in [TWS04] to eliminate some of the candidates. Subsequently, we retrieve the missing scores of the remaining candidates. Possibly in an iterative manner to be able to update the *min-k* estimate and the $\delta(d)$ values to allow further pruning. We stop when there are no candidates left.

Using this technique we can directly apply the probabilistic guarantees from [TWS04]. Theobald et al. introduce a new pruning technique to eliminate unpromising candidates. The idea is to estimate the probability that an item can get into the top-k list. If this probability is smaller than ε the algorithms prunes the candidate. More precisely, an item d is removed from the candidate list if

$$P[\sum\{s_i | i \notin E(d)\} > \delta(d)] < \varepsilon$$

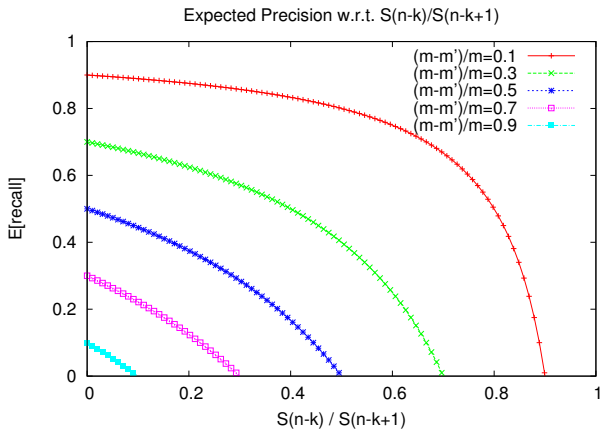

Figure 7.2: Expected relative recall w.r.t. the difference of the scores at rank k and rank $k+1$, for different values of $(m-m')/m$.

This pruning technique can cause errors in the final result quality, since it might happen that an item, that would get into the final top-k list, will be pruned. However, Theobald et al. [TWS04] give guarantees for the top-k results:

$$E[precision] = \sum_{r=0}^{k} P[precision = r/k] * r/k \geq (1-\varepsilon)$$

The application of this method after phase 2 in the algorithms that we consider in this thesis is straight forward, and the probabilistic guarantees from [TWS04] hold.

Chapter 8

Minerva ∞

Top-k aggregation queries, as presented in the previous Chapters 3, 4 and 6, assume that complete index-lists are stored at usually different servers, i.e. the data placement is given a-priori. As mentioned earlier, one could implement a distributed Web search engine based on this architecture but the resulting system would have serious scalability problems since the peers responsible for the most popular index-lists would have to cope with a high number of incoming requests.

Thus, the crucial challenge in developing successful P2P Web search engines is based on reconciling the following high-level, conflicting goals: on the one hand, to respond to user search queries with high quality results with respect to precision/recall, by employing an efficient distributed top-k query algorithm, and, on the other hand, to provide an infrastructure ensuring scalability and efficiency in the presence of a very large peer population and the very large amounts of data that must be communicated in order to meet the first goal.

Achieving ultra scalability is based on precluding the formation of central points of control during the processing of search queries. This dictates a solution that is highly distributed in both the data and computational dimensions. Such a solution leads to facilitating a large number of nodes pulling together their computational (storage, processing, and communication) resources, in essence increasing the total resources available for processing queries. At the same time, great care must be exercised in order to ensure efficiency of operation; that is, ensure that engaging greater numbers of peers does not lead to unnecessary high costs in terms of query response times, bandwidth requirements, and local peer work.

This chapter is based on our own work in [MTW05b] and presents Minerva∞, a P2P web search engine architecture, detailing its key design features, algorithms, and implementation. Minerva∞ features offer an infrastructure capable of attaining our scalability and efficiency goals. We report on a detailed experimental performance study of our implemented engine using real-world, web-crawled data collections and queries, which showcases our engine's efficiency and scalability. To the authors' knowledge, this is the first work that offers a

highly distributed (in both the data dimension and the computational dimension), scalable and efficient solution toward the development of internet-scale search engines.

8.1 Design Overview and Rationale

The fundamental distinguishing feature of Minerva∞ is its high distribution both in the data and computational dimensions. $Minerva\infty$ goes far beyond the state of the art in distributed top-k query processing algorithms, which are based on having nodes storing complete index lists for terms and running coordinator-based top-k algorithms [CW04, MTW05a]. From a data point of view, the principle is that the data items needed by top-k queries are the triplets $(term, docID, score)$ for each queried term (and not the index lists containing them). A proper distributed design for such systems then should appropriately distribute these items controllably so to meet the goals of scalability and efficiency. So data distribution in Minerva∞ is at the level of this, much finer data grain. From a system's point of view, the design principle we follow is to organize the key computations to engage several different nodes, with each node having to perform small (sub)tasks, as opposed to assigning single large task to a single node. These design choices we believe will greatly boost scalability (especially under skewed accesses).

Our approach to materializing this design relies on the employment of the novel notion of Term Index Networks (TINs). TINs may be formed for every term in our system, and they serve two roles: First, as an abstraction, encapsulating the information specific to a term of interest, and second, as a physical manifestation of a distributed repository of the term-specific data items, facilitating their efficient and scalable retrieval. A TIN can be conceptualized as a *virtual* node storing a *virtually global* index list for a term, which is constructed by the sorted merging of the separate complete index lists for the term computed at different nodes. Thus, TINs are comprised of nodes which collectively store different horizontal partitions of this global index list. In practice, we expect TINs to be employed only for the most popular terms (a few hundred to a few thousand) whose accesses are expected to form scalability and performance bottlenecks.

We will exploit the underlying network $G's$ architecture and related algorithms (e.g., for routing/lookup) to efficiently and scalably create and maintain TINs and for retrieving TIN data items, from any node of G. In general, TINs may form separate overlay networks, coexisting with the global overlay G. In practice, it may not always be necessary or advisable to form full-fledged separate overlays for TINs; instead, TINs will be formed as straightforward extensions of G: in this case, when a node n of G joins a TIN, only two additional links are added to the state of n linking it to its successor and predecessor nodes in the TIN. In this case, a TIN is simply a (circular) doubly-linked list.

The Minerva∞ algorithms are heavily influenced by the way the well-known, efficient top-k query processing algorithms (e.g., [FLN03]) operate, looking for

docIDs within certain ranges of score values. Thus, the networks' *lookup*(s) function, will be used using scores s as input, to locate the nodes storing docIDs with scores s.

A key point to stress here, however, is that top-k queries $Q(\{t_1, ..., r_r\}, k)$ can originate from any peer node p of G, which in general is not a member of any $I(t_i)$, $i = 1, ..., r$ and thus p does not have, nor can it easily acquire, the necessary routing state needed to forward the query to the TINs for the query terms. Our infrastructure, solves this by utilizing for each TIN a fairly small number (relative to the total number of data items for a term) of nodes of G which will be readily identifiable and accessible from any node of G and can act as *gateways* between G and this TIN, being members of both networks.

Finally, in order for any highly distributed solution to be efficient, it is crucial to keep as low as possible the time and bandwidth overheads involved in the required communication between the various nodes. This is particularly challenging for solutions built over very large scale infrastructures. To achieve this, the algorithms of Minerva∞ follow the principles put forward by top-performing, resource-efficient top-k query processing algorithms in traditional environments. Specifically, the principles behind favoring sequential index-list accesses to random accesses (in order to avoid high-cost random disk IOs) have been adapted in our distributed algorithms to ensure that: (i) sequential accesses of the items in the *global, virtual* index list dominate, (ii) they require either no communication, or at most a one-hop communication between nodes, and (iii) random accesses require at most $O(log|N|)$ messages.

To ensure the at-most-one-hop communication requirement for successive sequential accesses of TIN data, the Minerva∞ algorithms utilize an *order preserving hash function*, first proposed for supporting range queries in DHT-based data networks in [TP03]. An order preserving hash function $h_{op}()$ has the property that for any two values v_1, v_2, if $v_1 > v_2$ then $h_{op}(v_1) > h_{op}(v_2)$. This guarantees that data items corresponding to successive score values of a term t are placed either at the same or at neighboring nodes of $I(t)$. Alternatively, similar functionality can be provided by employing for each $I(t)$ an overlay based on skip graphs or skip nets [AS03, HJS$^+$03]. Since both order preserving hashing and skip graphs incur the danger for load imbalances when assigning data items to TIN nodes, given the expected data skew of scores, load balancing solutions are needed.

The design outlined so far thus leverages DHT technology to facilitate efficiency and scalability in key aspects of the system's operation. Specifically, posting (and deleting) data items for a term from any node can be done in $O(log|N|)$ time, in terms of the number of messages. Similarly, during top-k query processing, the TINs of the terms in the query can be also reached in $O(log|N|)$ messages. Furthermore, no single node is over-burdened with tasks which can either require more resources than available, or exhaust its resources, or even stress its resources for longer periods of time. In addition, as the top-k algorithm is processing different data items for each queried term, this involves gradually different nodes from each TIN, producing a highly distributed, scal-

able solution.

8.2 The Model

In general, we envision a widely distributed system, comprised of great numbers of peers, forming a collection with great aggregate computing, communication, and storage capabilities. Our challenge is to fully exploit these resources in order to develop an ultra scalable, efficient, internet-content search engine.

We expect that nodes will be conducting independent web crawls, discovering *documents* and computing *scores* of documents, with each score reflecting a document's importance with respect to *terms* of interest. The result of such activities is the formation of *index lists*, one for each term, containing relevant documents and their score for a term. More formally, our network consists of a set of nodes N, collectively storing a set D of documents, with each document having a unique identifier $docID$, drawn from a sufficiently large name space (e.g., 160 bits long). Set T refers to the set of terms. The notation $|S|$ denotes the cardinality of set S. The basic data items in our model are triplets of the form *(term, docID, score)*. In general, nodes employ some function $score(d,t) : D \rightarrow (0,1]$, which for some term t, produces the score for document d. Typically, such a scoring function utilizes tf*idf style statistical metadata.

The model is based on the following two fundamental operations. The $Post(t,d,s)$ operation, with $t \in T$, $d \in D$, and $s \in (0,1]$, is responsible for identifying a network node where the (t,d,s) triplet will be stored and storing it there. The operation $Query(T_i,k) : return(L_k)$, with $T_i \subseteq T$, k an integer, and $L_k = \{(d, TotalScore(d)) : d \in D,\ TotalScore(d) \geq \textit{min-k}\}$, is a top-k query operation. $TotalScore(d)$ denotes the aggregate score for d with respect to terms in T_i. Although there are several possibilities for the monotonic aggregate function to be used, we employ summation, for simplicity. Hence, $TotalScore(d) = \sum_{t \in T_i} score(d,t)$. For a given term, *min-k* refers to the k-th highest TotalScore, s_{min} (s_{max}) refers to the minimum (maximum) score value, and, given a score s, $next(s)$ ($prev(s)$) refers to the score value immediately following (preceding) s.

All nodes are connected on a *global* network G. G is an *overlay network*, modeled as a graph $G = (N,E)$, where E denotes the communication links connecting the nodes. E is explicitly defined by the choice of overlay network; for instance, for Chord, E consists of the successor, predecessor, and finger table (i.e., routing table) links of each node.

In addition to the global network G, encompassing all nodes, our model employs term-specific overlays, coined *Term Index Networks (TINs)*. $I(t)$ denotes the TIN for term t and is used to store and maintain all (t,d,s) items. TIN $I(t)$ is defined as $I(t) = (N(t), E(t))$, $N(t) \subseteq N$. Note that nodes in $N(t)$ have in addition to the links for participating in G, links needed to connect them to the $I(t)$ network. The model itself is independent of any particular overlay architecture.

$I(t).n(s_i)$ defines the node responsible for storing all triplets (t,d,s) for which

$score(d, t) = s = s_i$. When the context is well understood, the same node is simply denoted as $n(s)$.

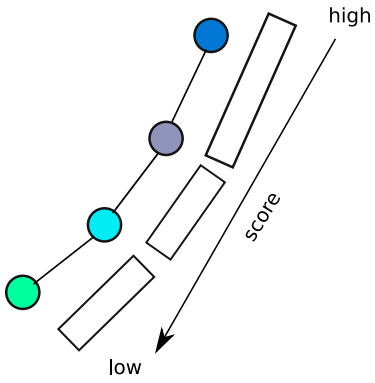

Figure 8.1: Illustration of an index-list that is distributed over multiple peers.

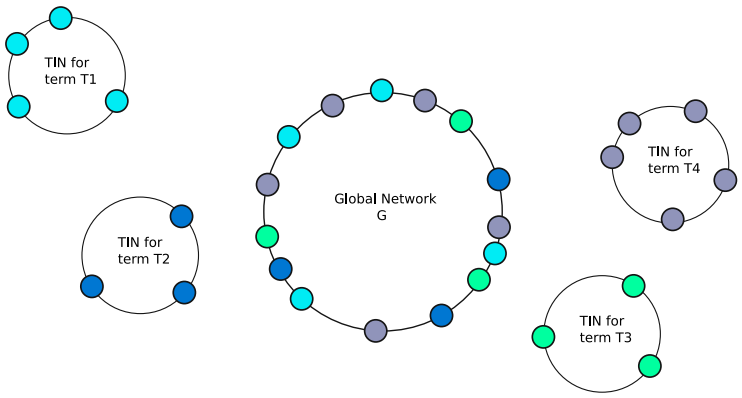

Figure 8.2: TINS and the global network G

8.3 Term Index Networks

In this section we describe and analyze the algorithms for creating TINs and populating them with data and nodes.

8.3.1 Beacons for Bootstrapping TINs

The creation of a TIN has these basic elements: posting data items, inserting nodes, and maintaining the connectivity of nodes to ensure the efficiency/scalability properties promised by the TIN overlay.

As mentioned, a key issue to note is that any node p in G may need to post (t, d, s) items for a term t. Since, in general, p is not a member of $I(t)$ and does

not necessarily know members of $I(t)$, efficiently and scalably posting items to $I(t)$ from any p becomes non-trivial. To overcome this, a bootstrapping process for $I(t)$ is employed which initializes an TIN $I(t)$ for term t. The basic novelty lies in the special role to be played by nodes coined *beacons*, which in essence become gateways, allowing the flow of data and requests between the G and $I(t)$ networks.

In the bootstrap algorithm, a predefined number of "dummy" items of the form $(t, \star, s_i)$ is generated in sequence for a set of predefined score values s_i, $i = 1, ..., u$. Each such item will be associated with a node n in G, where it will be stored. Finally, this node n of G will also be made a member of $I(t)$ by randomly choosing a previously inserted beacon node (i.e., for the one associated with an already inserted score value s_j, $1 \leq j \leq i-1$) as a gateway.

The following algorithm details the pseudo code for bootstrapping $I(t)$. It utilizes an order-preserving hash function $h_{op}() : T \times (0, 1] \rightarrow [m]$, where m is the size of the identifiers in bits and $[m]$ denotes the name space used for the overlay (e.g., all 2^{160} ids, for 160-bit identifiers). In addition, a standard hash function $h() : (0, 1] \rightarrow [m]$, (e.g. SHA-1) is used. The particulars of the order preserving hash function to be employed will be detailed after the presentation of the query processing algorithms which they affect.

Algorithm 8.1 Bootstrap $I(t)$

1: **input:** u: the number of "dummy" items $(t, \star, s_i)$, $i = 1, ..., u$
2: **input:** t: the term for which the TIN is created
3: $p = 1/u$
4: **for** $i = 1$ to u **do**
5: $s = i \times p$
6: $lookup(n.s) = h_{op}(t, s)$ { $n.s$ in G will become the next beacon node of $I(t)$ }
7: **if** $s = p$ **then**
8: $N(t) = \{n.s\}$
9: $E(t) = \emptyset$ {Initialized $I(t)$ with $n.s$ with the first dummy item}
10: **end if**
11: **if** $s \neq p$ **then**
12: $n_1 = h_{op}(t, s-p)$ {insert $n(s)$ into $I(t)$ using node $n(s-p)$ as gateway}
13: call $join(I(t), n_1, s)$
14: **end if**
15: store $(t, \star, s)$ at $I(t).n(s)$
16: **end for**

The bootstrap algorithm selects u "dummy" score values, i/u, $i = 1, ..., u$, finds for each such score value the node n in G where it should be placed (using $h_{op}()$), stores this score there and inserts n into the $I(t)$ network as well. At first, the $I(t)$ network contains only the node with the dummy item with score zero. At each iteration, another node of n is added to $I(t)$ using as gateway the node of G which was added in the previous iteration to $I(t)$. For simplicity of

presentation, the latter node can be found by simply hashing for the previous dummy value. A better choice for distributing the load among the beacons is to select at random one of the previously-inserted beacons and use it as a gateway.

Obviously, a single beacon per TIN suffices. The number u of beacon scores is intended to introduce a number of gateways between G and $I(t)$ so to avoid potential bottlenecks during TIN creation. u will typically be a fairly small number so the total beacon-related overhead involved in the TIN creation will be kept small. Further, we emphasize that beacons are utilized by the algorithm posting items to TINs. Post operations will in general be very rare compared to query operations and query processing does not involve the use of beacons.

Finally, note that the algorithm uses a $join()$ routine that adds a node $n(s)$ storing score s into $I(t)$ using a node n_1 known to be in $I(t)$ and thus, has the required state. The new node $n(s)$ must occupy a position in $I(t)$ specified by the value of $h_{op}(t, s)$. Note that this is ensured by using $h(nodeID)$, as is typically done in DHTs, since these node IDs were selected from the order-preserving hash function. Besides the side-effect of ensuring the order-preserving position for the nodes added to a TIN, the join routine is otherwise straightforward: if the TIN is a full-fledged DHT overlay, join() is updating the predecessor/successor pointers, the $O(log|N|)$ routing state of the new node, and the routing state of each $I(t)$ node pointing to it, as dictated by the relevant DHT algorithm. If the TIN is simply a doubly-linked list, then only predecessor/successor pointers are the new node and its neighbors are adjusted.

8.3.2 Posting Data to TINs

The posting of data items is now made possible using the bootstrapped TINs. Any node n_1 of G wishing to post an item (t, d, s) first locates an appropriate node of G, n_2 that will store this item. Subsequently it inserts node n_2 into $I(t)$. To do this, it randomly selects a beacon score and associated beacon node, from all available beacons. This is straightforward given the predefined beacon score values and the hashing functions used. The chosen beacon node has been made a member of $I(t)$ during bootstrapping. Thus, it can "escort" n_2 into $I(t)$.

The following provides the pseudo code for the posting algorithm.

Algorithm 8.2 Posting Data to $I(t)$

1: **input:** t, d, s: the item to be inserted by a node n_1
2: $n(s) = h_{op}(t, s)$
3: n_1 sends (t, d, s) to $n(s)$
4: **if** $n(s) \notin N(t)$ **then**
5: $n(s)$ selects randomly a beacon score s_b
6: $lookup(n_b) = h_{op}(t, s_b)$ { n_b is the beacon node storing beacon score s_b }
7: $n(s)$ calls $join(I(t), n_b, s)$
8: **end if**
9: store $((t, d, s)$

By design, the post algorithm results in a data placement which introduces

two characteristics, that will be crucial in ensuring efficient query processing. First, (as the bootstrap algorithm does) the post algorithm utilizes the order-preserving hash function. As a result, any two data items with consecutive score values for the same term will be placed by definition in nodes of G which will become one-hop neighbors in the TIN for the term, using the join() function explained earlier. Note, that within each TIN, there are no "holes". A node n becomes a member of a TIN network if and only if a data item was posted, with the score value for this item hashing to n. It is instructing here to emphasize that if TINs were not formed and instead only the global network was present, in general, any two successive score values could be falling in nodes which in G could be many hops apart. With TINs, following successor (or predecessor) links always leads to nodes where the next (or previous) segment of scores have been placed. This feature in essence ensures the at-most-one-hop communication requirement when accessing items with successive scores in the global virtual index list for a term.

Second, the nodes of any $I(t)$ become responsible for storing specific segments (horizontal partitions) of the global virtual index list for t. In particular, an $I(t)$ node stores all items for t for a specific (range of) score value, posted by any node of the underlying network G.

8.3.3 Complexity Analysis

The bootstrapping $I(t)$ algorithm is responsible for inserting u beacon items. For each beacon item score, the node $n.s$ is located by applying the $h_{op}()$ function and routing the request to that node (step 6). This will be done using G's lookup algorithm in $O(log|N|)$ messages. The next key step is to locate the previously inserted beacon node (step 12) (or any beacon node at random) and sending it the request to join the TIN. Step 12 again involves $O(log|N|)$ messages. The actual join() routine will cost $O(log^2|N(t)|)$ messages, which is the standard join() message complexity for any DHT of size $N(t)$. Therefore, the total cost is $O(u \times (log|N| + log^2|N(t)|)$ messages.

The analysis for the posting algorithm is very similar. For each $post(t, d, s)$ operation, the node n where this data item should be stored is located and the request is routed to it, costing $O(log|N|)$ messages (step 2). Then a random beacon node is located, costing $O(log|N|)$ messages, and then the join() routine is called from this node, costing $O(log^2|N(t)|)$ messages. Thus, each post operation has a complexity of $O(log|N|) + O(log^2|N(t)|)$ messages.

Note that both of the above analysis assumed that each $I(t)$ is a full-blown DHT overlay. This permits a node to randomly select any beacon node to use to join the TIN. Alternatively, if each $I(t)$ is simply a (circular) doubly-linked list, then a node can join a TIN using the beacon storing the beacon value that is immediately preceding the posted score value. This requires $O(log|N|)$ hops to locate this beacon node. However, since in this case the routing state for each node of a TIN consists of only the two (predecessor and successor) links, the cost to join is in the worst case $O(|N(t)|$, since after locating the beacon node with

the previous beacon value, $O(|N(t)|$ successor pointers may need to be followed in order to place the node in its proper order-preserving position. Thus, when TINs are simple doubly-linked lists, the complexity of both the bootstrap and post algorithms are $O(log|N| + |N(t)|)$ messages.

8.4 Load Balancing

8.4.1 Order-Preserving Hashing

The order preserving hash function to be employed is important for several reasons. First, for simplicity, the function can be based on a simple linear transform. Consider hashing a value $f(s) : (0, 1] \rightarrow I$, where $f(s)$ transforms a score s into an integer; for instance, $f(s) = 10^6 \times s$. Function $h_{op}()$ can be defined then as

$$h_{op}(s) = \frac{f(s) - f(s_{min})}{f(s_{max}) - f(s_{min})} \times 2^m \tag{8.1}$$

Although such a function is clearly order-preserving, it has the drawback that it produces the same output for items of equal scores of different terms. This leads to the same node storing for all terms all items having the same score. This is undesirable since it cannot utilize all available resources (i.e., utilize different sets of nodes to store items for different terms). To avoid this, $h_{op}()$ is refined to take as input the term name, which provides the necessary functionality, as follows.

$$h_{op}(t, s) = (h(t) + \frac{f(s) - f(s_{min})}{f(s_{max}) - f(s_{min})} \times 2^m) \ mod \ 2^m \tag{8.2}$$

The term $h(t)$ adds a different random offset for different terms, initiating the search for positions of term score values at different, random, offsets within the namespace. Thus, by using the $h(t)$ term in $h_{op}(t, s)$ the result is that any data items having equal scores but for different terms are expected to be stored at different nodes of G.

Another benefit stems from ameliorating the storage load imbalances that result from the non-uniform distribution of score values. Assuming a uniform placement of nodes in G, the expected non-uniform distribution of scores will result in a non-uniform assignment of scores to nodes. Thus, when viewed from the perspective of a single term t, the nodes of $I(t)$ will exhibit possibly severe storage load imbalances. However, assuming the existence of large numbers of terms (e.g., a few thousand), and thus data items being posted for all these terms over the same set of nodes in G, given the randomly selected starting offsets for the placement of items, it is expected that the severe load imbalances will disappear. Intuitively, overburdened nodes for the items of one term are expected to be less burdened for the items of other terms and vice versa.

But even with the above hash function, very skewed score distributions will lead to storage load imbalances.

A related, more subtle problem is that, especially for the top score values which belong to the score value space that is sparse, the $I(t)$ nodes responsible for storing them will be storing only a single or a very small number of items. Thus, during top-k query processing, in order to retrieve enough items for each term, too many hops will be necessary.

Expecting that exponential-like distributions of score values will appear frequently, we developed a hash function that is order-preserving and handles load imbalances by assigning score segments of exponentially decreasing sizes to an exponentially increasing number of nodes (cf. Figure 8.3). For instance, the sparse top 1/2 of the scores distribution is to be assigned to a single node, the next 1/4 of scores is to be assigned to 2 nodes, the next 1/8 of scores to 4 nodes, etc.

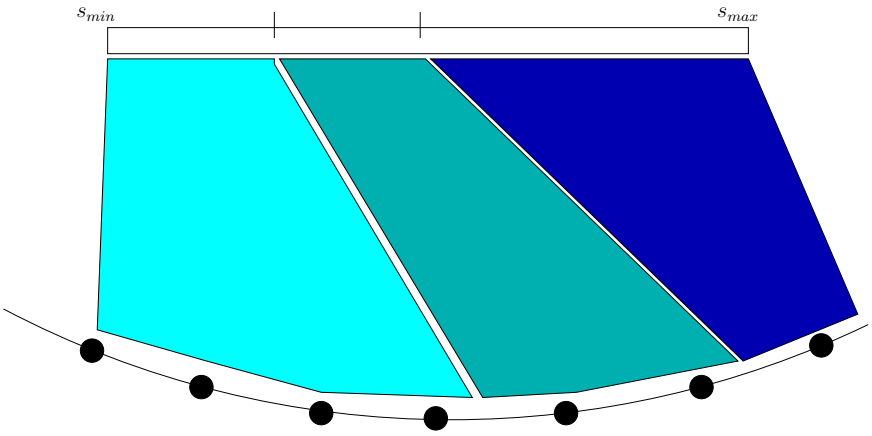

Figure 8.3: Illustration of the load-balancing, order-preserving hash function. The first part (sparse since high score area) is assigned to one peer. The next half is assigned to 2 nodes....

First, we define $\sigma(s)$ as the segment where score s belongs to be,

$$\sigma(s) = \lceil log(f(s_{max})) \rceil - \lceil log(f(s)) \rceil \tag{8.3}$$

where the size of a segment $\varsigma(\sigma(s))$ is given by,

$$\varsigma(\sigma(s)) = 2^{\sigma(s)} \tag{8.4}$$

Next we define the boundaries of the score range for each segment as follows,

$$f(s_{min}^{\sigma(s)}) = \frac{f(s_{max})}{2^{\sigma(s)+1}} \tag{8.5}$$

$$f(s_{max}^{\sigma(s)}) = \frac{f(s_{max})}{2^{\sigma(s)}} \tag{8.6}$$

Last, we define the offset, $\phi(\sigma(s))$, that is, the starting node where the scores of a given score segment will be placed as follows,

$$\phi(\sigma(s)) = 2^{\sigma(s)} - 1 \tag{8.7}$$

Combining the above, yields the order-preserving hash function that solves the problem of load imbalances for exponential-like score distributions, defined as

$$h_{op}(t,s) = h(t) + (\phi(\sigma(s)) + (\frac{f(s_{max}^{\sigma(s)}) - f(s)}{f(s_{max}^{\sigma(s)}) - f(s_{min}^{\sigma(s)})} \times \varsigma(\sigma(s)) \ mod \ 2^{\rho} \tag{8.8}$$

Thus, the top half of the score domain (0.5, 1] is assigned to the top segment which has a size of one node; the next segment of scores in (0.25, 0.5] is assigned to the second-top segment having 2 nodes, etc.

The output of the hash is taken *mod* 2^{ρ}. This value is meant to put an upper bound on the number of nodes which can be members of a TIN. Thus, each global virtual index list is not distributed over all nodes, but over a much smaller segment of the ID space of G. This avoid having each node being a member of (almost) all index lists, participating into all TIN overlays, storing and maintaining routing state per each TIN, which obviously does not scale. In the experimentation section we shall see the effect of this hash function on load balancing, posting data items from a large number of real-world long index lists.

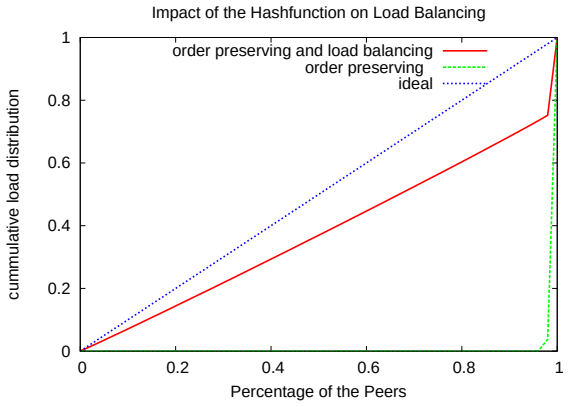

Figure 8.4: The impact of hash function on the load imbalances

As for a measure of load imbalances we consider measure the Gini coefficient of the load distribution, that is defined as

$$G = 1 - 2\int_0^1 L(x)dx$$

where L(x) is the Lorenz curve of the underlying distribution.

Figure 8.4 shows an example of the impact of hash function on the overall load balancing performance. Here we consider the load of peers sharing the all .GOV index lists, i.e. the index lists for all terms that occur in one of the .GOV queries. Figure 8.4 displays the observed Lorenz curve and we can measure the Gini coefficients for both hash functions: The standard order preserving hash function causes an coefficient of 0.97, indicating extremely large load imbalances. With our proposed method we are able to reach a coefficient of 0.61 that is pretty good considering the fact that there are no complex data placement algorithms involved. In the following section we will see how this initial data placement can be further improved by using a data migration algorithm.

8.4.2 TIN Data Migration

Exploiting the key characteristics of our data, Minerva∞ can ensure further load balancing with small overheads. Specifically, index lists data entries are small in size and are very rarely posted and/or updated. In this subsection we outline our approach for improved load balancing.

We require that each peer posting index list entries, first computes a (equi-width) histogram of its data with respect to its score distribution. Assuming a targeted $|N(t)|$ number of nodes for the TIN of term t, it can create $|N(t)|$ equal-size partitions, with $lowscore_i, highscore_i$ denoting the score ranges associated with partition i, $i = 1, ..., |N(t)|$. Then it can simply utilize the posting algorithm shown earlier, posting using the $lowscore_i$ scores for each partition. The only exception to the previously shown post algorithm is that the posting peer now posts at each iteration a complete partition of its index list, instead of just a single entry.

The above obviously can guarantee perfect load balancing. However, subsequent postings (typically by other peers) may create imbalances, since different index lists may have different score distributions. Additionally, when ensuring overall load balancing over multiple index lists being posting by several peers, the order-preserving property of the placement must be guaranteed. Our approach for solving these problems is as follows. First, again the posting peer is required to compute a histogram of its index list. Second, the histogram of the TIN data (that is, the entries already posted) is stored at easily identifiable nodes. Third, the posting peer is required to retrieve this histogram and 'merge' it with his own. Fourth, the same peer identifies how the total data must now be split into $|N(t)|$, equal-size partitions of consecutive scores. Finally, it identifies all data movements (from TIN peer to TIN peer) necessary to redistribute the total TIN data so that load balancing and order preservation is ensured.

Detailed presentation of the possible algorithms for this last step and their respective comparison is beyond the scope of this work. We simply mention that total TIN data sizes is expected to be very small (in actual number of bytes stored and moved). For example, even with several dozens of peers posting different, even large, multi-million-entry, index lists, in total the complete TIN

data size will be a few hundred MBs, creating a total data transfer movement equivalent to that of downloading a few dozen MP3 files. Further, index lists' data posting to TINs is expected to be a very infrequent operation (compared to search queries). As a result, ensuring load balancing across TIN nodes proves to be relative inexpensive.

The approaches to index lists' data posting outlined above can be used competitively or even be combined. When posting index lists with exponential score distributions, by design the posting of data using the order-preserving hash function, will be adequately load balanced and nothing else is required. Conversely, when histogram information is available and can be computed by posting peers, the TIN data migration approach will yield load balanced data placement.

A more subtle issue is that posting with the order-preserving hash function also facilitates random accesses of the TIN data, based on random score values. That is, by hashing for any score, we can find the TIN node holding the entries with this score. This becomes essential if the web search engine is to employ top-k query algorithms which are based on random accesses of scores. In our work, our top-k algorithms avoid random accesses, by design. However, the above point should be kept in mind since there are recently-proposed distributed top-k algorithms, relying on random accesses and more efficient algorithms may be proposed in the future.

8.5 Top-k Query Processing

The algorithms in this section focus on how to exploit the infrastructure presented previously in order to efficiently process top-k queries. The main efficiency metrics are query response times and network bandwidth requirements.

8.5.1 The Basic Algorithm

Consider a top-k query of the form $Q(\{t_1, ..., t_r\}, k)$ involving r terms that is generated at some node n_{init} of G. Query processing is based on the following ideas. It proceeds in phases, with each phase involving 'vertical' and 'horizontal' communication between the nodes within TINs and across TINs, respectively. The vertical communications between the nodes of a TIN are occurring in parallel across all r TINs named in the query, gathering a threshold number of data items from each term. There is a moving coordinator node, that will be gathering the data items from all r TINs that enable it to compute estimates of the top-k result. Intermediate estimates of the top-k list will be passed around, as the coordinator role moves from node to node in the next phase where the gathering of more data items and the computation of the next top-k result estimate will be computed.

The presentation shows separately the behavior of the query initiator, the (moving) query coordinator, and the TIN nodes.

Query Initiator

The initiator calculates the set of *start nodes*, one for each term, where the query processing will start within each TIN. Also, it randomly selects one of the nodes (for one of the TINs) to be the initial coordinator. Finally, it passes on the query and the coordinator ID to each of the start nodes, to initiate the parallel vertical processing within TINs.

The following pseudo code details the behavior of the initiator.

Algorithm 8.3 Top-k QP: Query Initiation at node $G.n_{init}$

1: **input:** Given query $Q = \{t_1,..,t_r\}, k$:
2: **for** $i = 1$ to r **do**
3: $startNode_i = I(t_i).n(s_{max}) = h_{op}(t_i, s_{max})$
4: **end for**
5: Randomly select c from $[1, ..., r]$
6: $coordID = I(t_c).n(s_{max})$
7: **for** $i = 1$ to r **do**
8: send to $startNode_i$ the data $(Q, coordID)$
9: **end for**

Processing Within each TIN

Processing within a TIN is always initiated by the start node. There is one start node per communication phase of the query processing. In the first phase, the start node is the top node in the TIN which receives the query processing request from the initiator. The start node then starts the gathering of data items for the term by contacting enough nodes, following successor links, until a threshold number γ (that is, a batch size) of items has been accumulated and sent to the coordinator, along with an indication of the maximum score for this term which has not been collected yet, which is actually either a locally stored score or the maximum score of the next successor node. The latter information is critical for the coordinator in order to intelligently decide when the top-k result list has been computed and terminate the search. In addition, each start node sends to the coordinator the ID of the node of this TIN to be the next start node, which is simply the next successor node of the last accessed node of the TIN. Processing within this TIN will be continued at the new start node when it receives the next message from the coordinator starting the next data-gathering phase.

Algorithm 8.4 presents the pseudo code for TIN processing.

Recall, that because of the manner with which items and nodes have been placed in a TIN, following succ() links, items are collected starting from the item with the highest score posted for this term and proceeding in sorted descending order based on scores.

Algorithm 8.4 Top-k QP: Processing by a start node within a TIN

```
1: input: A message either from the initiator or the coordinator
2: tCollection_i = ∅
3: n = startNode_i
4: while |tCollection_i| < γ do
5:    while |tCollection_i| < γ AND more items exist locally do
6:       define the set of local items L = {(t_i, d, s) in n}
7:       send to coordID : L
8:       |tCollection_i| = |tCollection_i| + |L|
9:    end while
10:   n = succ(n)
11: end while
12: bound_i = max score stored at node n
13: send to coordID : n and bound_i
```

Moving Query Coordinator

Initially, the coordinator is randomly chosen by the initiator to be one of the original start nodes. First, the coordinator uses the received collections and runs a version of the *NRA* top-k processing algorithm, locally producing an estimate of the top-k result. As is also the case with classical top-k algorithms, the exact result is not available at this stage since only a portion of the required information is available. Specifically, some documents with high enough TotalScore to qualify for the top-k result are still missing. Additionally, some documents may also be seen in only a subset of the collections received from the TINs so far, and thus some of their scores are missing, yielding only a partially known TotalScore.

A key to the efficiency of the overall query processing process is the ability to prune the search and terminate the algorithm even in the presence of missing documents and missing scores. To do this, the coordinator first computes an estimate of the top-k result, which includes only documents whose TotalScores are completely known, defining the *min-k* value (i.e. the smallest score in the top-k list estimate). Then, it utilizes the $bound_i$ values received from each start node. When a score for a document d is missing for term i, it can be replaced with $bound_i$ to estimate the $TotalScore(d)$. This is done for all such d with missing scores. If *min-k* $> TotalScore(d)$ for all d with missing scores then there is no need to continue the process for finding the missing scores, since the associated documents could never belong to the top-k result. Similarly, if *min-k* $> \sum_{i=1,\ldots,r} bound_i$, then similarly there is no need to try to find any other documents, since they could never belong to the top-k result. When both of these conditions hold, the coordinator terminates the query processing and returns the top-k result to the initiator.

If the processing must continue, the coordinator starts the next phase, sending a message to the new start node for each term, whose ID was received in the message containing the previous data collections. In this message the coordina-

tor also indicates the ID of the node which becomes the coordinator in this next phase. The next coordinator is defined to be the node in the same TIN as the previous coordinator whose data is to be collected next in the vertical processing in this TIN (i.e., the next start node at the coordinator's TIN). Alternatively, any other start node can be randomly chosen as the coordinator.

Algorithm 8.5 details the behavior of the coordinator.

Algorithm 8.5 Top-k QP: Coordination

1: **input:** For each i: $tCollection_i$ and $newstartNode_i$ and $bound_i$
2: $tCollection = \bigcup_i tCollection_i$
3: compute a (new) top-k list estimate using $tCollection$, and $min\text{-}k$
4: $candidates = \{d | d \notin$ top-k list$\}$
5: **for** all $d \in candidates$ **do**
6: $worstScore(d)$ is the partial TotalScore of d
7: $bestScore(d) := worstScore(d) + \sum_{j \in MT} bound_j$ {Where MT is the set of term ids with missing scores }
8: **if** $bestScore(d) < min\text{-}k$ **then**
9: remove d from $candidates$
10: **end if**
11: **end for**
12: **if** $candidates$ is empty **then**
13: exit()
14: **end if**
15: **if** $candidates$ is not empty **then**
16: $coordID_{new} = pred(n)$
17: calculate new size threshold γ
18: **for** $i = 1$ to r **do**
19: send to $startNode_i$ the data $(coordID_{new}, \gamma)$
20: **end for**
21: **end if**

8.5.2 Complexity Analysis

The overall complexity has three main components: the cost incurred for (i) the communication between the query initiator and the start nodes of the TINs, (ii) the vertical communication within a TIN, and (iii) the horizontal communication between the current coordinator and the current set of start nodes.

The query initiator needs to lookup the identity of the initial start nodes for each one of the r query terms and route to them the query and the chosen coordinator ID. Using the G network, this incurs a communication complexity of $O(r \times log|N|)$ messages. Denoting with *depth* the average (or maximum) number of nodes accessed during the vertical processing of TINs, overall $O(r \times depth)$ messages are incurred due to TIN processing, since subsequent accesses within a TIN require, by design, one-hop communication. Each horizontal communication in each phase of query processing between the coordinator and the r start

nodes requires $O(r \times log|N|)$ messages. Since such horizontal communication takes place at every phase, this yields a total of $O(phases \times r \times log|N|)$ messages. Hence, the total communication cost complexity is

$$cost = O(phases \times r \times log|N| + r \times log|N| + r \times depth) \tag{8.9}$$

This total cost is the worst case cost; we expect that the cost incurred in most cases will be much smaller, since horizontal communication across TINs can be much more efficient than $O(log|N|)$, as follows. The query initiator can first resolve the ID of the coordinator (by hashing and routing over G) and then determine its actual physical address (i.e., its IP address), which is then forwarded to each start node. In turn, each start node can forward this from successor to successor in its TIN. In this way, at any phase of query processing, the last node of a TIN visited during the vertical processing, can send the data collection to the coordinator using the coordinator's physical address. The current coordinator also knows the physical address of the next coordinator (since this was the last node visited in its own TIN from which it received a message with the data collection for its term) and of the next start node for all terms (since these are the last nodes visited during vertical processing of the TINs, from which it received a message). Thus, when sending the message to the next start nodes to continue vertical processing, the physical addresses can be used. The end result of this is that all horizontal communication requires one message, instead of $O(log|N|)$ messages. Hence, the total communication cost complexity now becomes

$$cost = O(phases \times r + r \times log|N| + r \times depth) \tag{8.10}$$

As nodes are expected to be joining and leaving the underlying overlay network G, occasionally, the physical addresses used to derive the cost of (8.10) will not be valid. In this case, the reported errors will lead to nodes using the high-level IDs instead of the physical addresses, in which case the cost is that given by (8.9).

8.6 Expediting Top-k Query Processing

In this section we develop optimizations that can further speedup the performance of top-k query processing. These optimizations are centered on: (i) the 'vertical' replication of term-specific data among the nodes of a TIN, and (ii) the 'horizontal' replication of data across TINs.

8.6.1 TIN Data Replication

There are two key characteristics of the data items in our model, which permit their large-scale replication. First, data items are rarely posted and even more rarely updated. Second, data items are very small in size (e.g. < 50 bytes each). Hence, replication protocols will not cost significantly either in terms of replica state maintenance, or in terms of storing the replicas.

Vertical Data Replication

The issue to be addressed here is how to appropriately replicate term data within TIN peers so to gain in efficiency. The basic structure of the query processing algorithm presented earlier facilitates the easy incorporation of a replication protocol into it. Recall, that in each TIN $I(t)$, query processing proceeds in phases, and in each phase a TIN node (the current start node) is responsible for visiting a number of other TIN nodes, a successor at a time, so that enough, i.e. a batch size of data items for t are collected. The last visited node in each phase which collects all data items, can initiate a 'reverse' vertical communication, in parallel to sending the collection to the coordinator. With this reverse vertical communication thread, each node in the reverse path sends to its predecessor only the data items its has not seen. In the end, all nodes in the path from the start node to the last node visited will eventually receive a copy of all items collected during this phase, storing locally the pair $(lowestscore, highestscore)$, marking its lowest and highest locally stored scores. Since this is straightforward, the pseudo code is omitted for space reasons.

Since a new posting involves all (or most) of the nodes in these paths, each node knows when to initiate a new replication to account for the new items.

Exploiting Replicas

The start node selected by the query initiator no longer needs to perform a successor-at-a-time traversal of TIN in the first phase, since the needed data (replicas are stored locally). However, vertical communication was also useful for producing the ID of the next start node for this TIN. A subtle point to note here is that the coordinator can itself determine the new start node for the next phase, even without receiving explicitly this ID at the end of vertical communication. This can simply be done using the minimum score value ($bound_i$) it has received for term t_i; the ID of the next start node is found hashing for score $prev(bound_i)$.

Additionally, the query initiator can select as start nodes the nodes responsible for storing a random (expected to be high score) and not always the maximum score, as it does up to now. Similarly, the coordinator when selecting the ID of the next start node for the next batch retrieval for a term, it can choose to hash for a score value that is lower than the score $prev(bound_i)$. Thus, random start nodes within a TIN are selected at different phases and these gather the next batch of data from the proper TIN nodes, using the TIN DHT infrastructure for efficiency. The details of how this is done, are omitted for space reasons.

Horizontal Data Replication

TIN data may also be replicated horizontally. The simplest strategy is to create replicated TINs for popular terms. This involves the posting of data into all TIN replicas. The same algorithms can be used as before for posting, except now when hashing, instead of using the term t as input to the hash function, each replica of t must be specified (eg $t.v$, where v stands for a version/replica

number). Again, the same algorithms can be used for processing queries, with the exception that each query can now select one of the replicas of $I(t)$, at random.

Overall, TIN data replication leads to savings in the number of messages and response time speedups. Furthermore, several nodes are offloaded since they no longer have to partake in the query processing process. With replication, therefore, overall the same number of nodes will be involved in processing a number of user queries, except that each query will be employing a smaller set of peers, yielding response time and bandwidth benefits. In essence, TIN data replication increases the efficiency of the engine, without adversely affecting its scalability. Finally, it should be stressed that such replication will also improve the availability of data items and thus replication is imperative. Indirectly, for the same reason the quality of the results with replication will be higher, since lost items inevitably lead to errors in the top-k result.

8.7 Experimentation

8.7.1 Experimental Testbed

Our implementation was written in Java. Experiments were performed on 3GHz Pentium PCs. Since deploying full-blown, large networks is not an option, we opted for simulating large numbers of nodes as separate processes on the same PC, executing the real Minerva∞ code. A 10,000 node network was simulated.

A real-world data collection was used in our experiments: GOV. The GOV collection consists of the data of the TREC-12 Web Track and contains roughly 1.25 million (mostly HTML and PDF) documents obtained from a crawl of the .gov Internet domain (with total index list size of 8 GB). The original 50 queries from the Web Track's distillation task were used. These are term queries, with each query containing up to 4 terms. The index lists contained the original document scores computed as tf * log idf. tf and idf were normalized by the maximum tf value of each document and the maximum idf value in the corpus, respectively. In addition, we employed an extended GOV (XGOV) setup, with a larger number of query terms and associated index lists. The original 50 queries were expanded by adding new terms from synonyms and glosses taken from the WordNet thesaurus (http://www.cogsci.princeton.edu/∼wn). The expansion yielded queries with, on average, twice as many terms, up to 18 terms.

8.7.2 Performance Tests and Metrics

Efficiency Experiments

The data (index list entries) for the terms to be queried were first posted. Then, the GOV/XGOV benchmark queries were executed in sequence. For simplicity, the query initiator node assumed the role of a fixed coordinator. The experiments used the following metrics:

Bandwidth. This shows the number of bytes transferred between all the nodes involved in processing the benchmarks' queries. The benchmarks' queries were grouped based on the number of terms they involved. In essence, this grouping created a number of smaller sub-benchmarks.

Query Response Time. This represents the elapsed, "wall-clock" time for running the benchmark queries. We report on the wall-clock times per sub-benchmark and for the whole GOV and XGOV benchmarks.

Hops. This reports the number of messages sent over our network infrastructures to process all queries. For communication over the global DHT G, the number of hops was set to be $log|N|$ (ie when the query initiator contacts the first set of start nodes for each TIN). Communication between peers within a TIN requires, by design, one hop at a time.

To avoid the overestimation of response times due to the competition between all processes for the PC's disk and network resources, and in order to produce reproducible and comparable results for tests ran at different times, we opted for simulating disk IO latency and network latency. Specifically, each random disk IO was modeled to incur a disk seek and rotational latency of 9 ms, plus a transfer delay dictated by a transfer rate of 8MB/s. For network latency we utilized typical round trip times (RTTs) of packets and transfer rates achieved for larger data transfers between widely distributed entities [SL00]. We assumed a RTT of 100 ms. When peers simply forward the query to a next peer, this is assumed to take roughly 1/3 of the RTT (since no ACKs are expected). When peers sent more data, the additional latency was dictated by a "large" data transfer rate of 800KBits/s, which includes the sender's uplink bandwidth, the receivers downlink bandwidth, and the average Internet bandwidth typically witnessed. This figure is the average throughput value measured (using one stream – one cpu machines) in experiments conducted for measuring wide area network throughput (sending 20MB files between SLAC nodes (Stanford's Linear Accelerator Centre) and nodes in Lyon France [SL00] using NLANR's iPerf tool [Tir03].

Scalability Experiments

The tested scenarios varied the query load to the system, measuring the overall time required to complete the processing of all queries in a queue of requests. Our experiments used a queue of identical queries involving four terms, with varying index lists characteristics. Two of these terms had small index lists (with over 22,000 and over 42,000 entries) and the other two lists had sizes of over 420,000 entries. For each query the (different) query initiating peer played the role of the coordinator.

The key here is to measure contention for resources and its limits on the possible parallelization of query processing. Each TIN peer uses his disk, his uplink bandwidth to forward the query to his TIN successor, and to send data to the coordinator. Uplink/downlink bandwidths were set to 256Kbps/1Mbps. Similarly, the query initiator utilizes its downlink bandwidth to receive the batches of data in each phase and its uplink bandwidth to send off the query to the next TIN

start nodes. These delays define the possible parallelization of query execution. By involving the two terms with the largest index lists in the queries, we ensured the worst possible parallelization (for our input data), since they induced the largest batch size, requiring the most expensive disk reads and communication.

8.7.3 Performance Results

Overall, each benchmark experiment required between 2 to 5 hours for its real-time execution, a big portion of which was used up by the posting procedure.

Figures 8.5, 8.6, 8.7 and 8.8,8.9,8.10 show the bandwidth, response times, and hops results for the GOV and XGOV group-query benchmarks. Note, that different query groups have in general mutually-incomparable results, since they involve different index lists with different characteristics (such as size, score distributions etc).

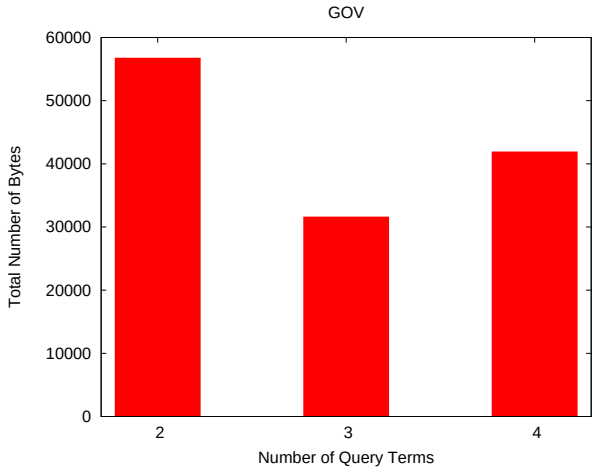

Figure 8.5: GOV Results: Bandwidth

The 2-term queries introduced the biggest overheads. There are 29 2-term, 7 3-term, and 4 4-term queries in GOV.

In XGOV the biggest overhead was introduced by the 8 7-term and 6 11-term queries. Table 8.1 shows the total benchmark execution times, network bandwidth consumption, as well as the number of hops for the GOV and XGOV benchmarks.

Benchmark	**Hops**	**Bandwidth(KB)**	**Time(s)**
GOV	22050	130189	2212
XGOV	146168	744700	10372

Table 8.1: Total GOV and XGOV Results

Generally, for each query, the number of terms and the size of the corre-

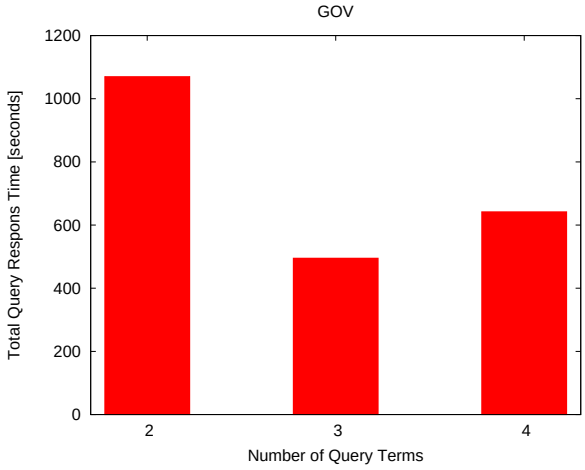

Figure 8.6: GOV Results: Execution Time

sponding index list data are the key factors. The central insight here is that the choice of the NRA algorithm was the most important contributor to the overhead. The adaptation of more efficient distributed top-k algorithms within Minerva∞ (such as our own [MTW05a], which also disallow random accesses) can reduce this overhead by one to two orders of magnitude. This is due to the fact that the top-k result can be produced without needing to delve deeply into the index lists' data, resulting in drastically fewer messages, bandwidth, and time requirements.

Figure 8.11 shows the scalability experiment results. Query loads tested represent queue sizes of 10, 100, 1000, and 10000 identical queries simultaneously arriving into the system. This figure also shows what the corresponding time would be if the parallelization contributed by the Minerva∞ architecture was not possible; this would be the case, for example, in all related-work P2P search architectures and also distributed top-k algorithms, where the complete index lists at least for one query term are stored completely at one peer.

The scalability results show the high scalability achievable with Minerva∞. It is due to the "pipelining" that is introduced within each TIN during query processing, where a query consumes small amounts of resources from each peer, pulling together the resources of all (or most) peers in the TIN for its processing. For comparison we also show the total execution time in an environment in which each complete index list was stored in a peer. This is the case for most related work on P2P search engines and on distributed top-k query algorithms. In this case, the resources of the single peer storing a complete index list are required for the processing of all communication phases and for all queries in the queue. In essence, this yields a total execution time that is equal to that of a sequential execution of all queries using the resources of the single peers storing the index lists for the query terms. Using this as a base comparison, Minerva∞ is shown

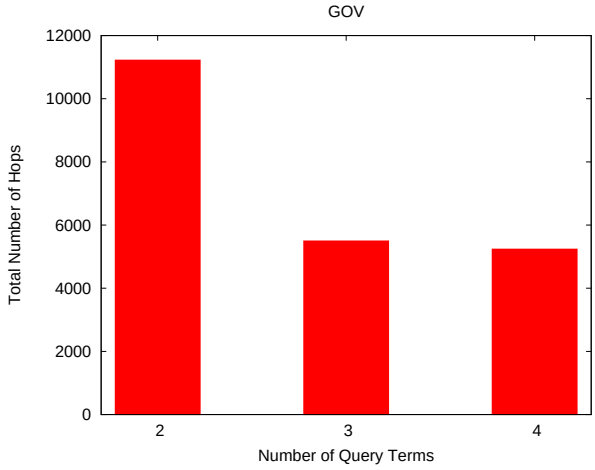

Figure 8.7: GOV Results: Hops

to enjoy approximately two orders of magnitude higher scalability. Since in our experiments there are approximately 100 nodes per TIN, this defines the maximum scalability gain.

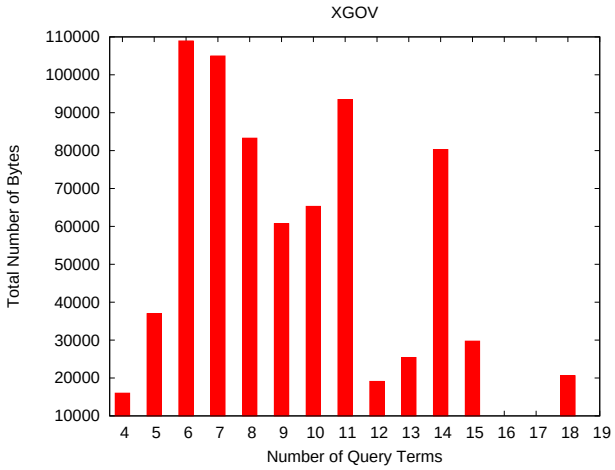

Figure 8.8: XGOV Results: Bandwidth

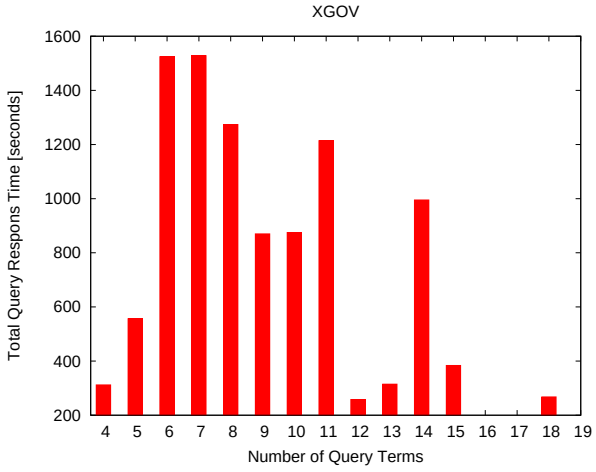

Figure 8.9: XGOV Results: Execution Time

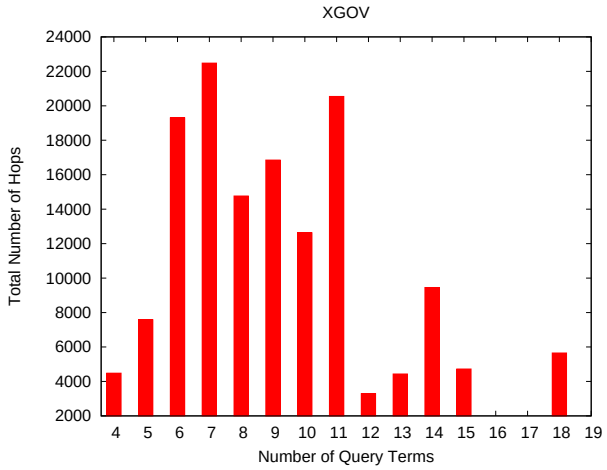

Figure 8.10: XGOV Results: Hops

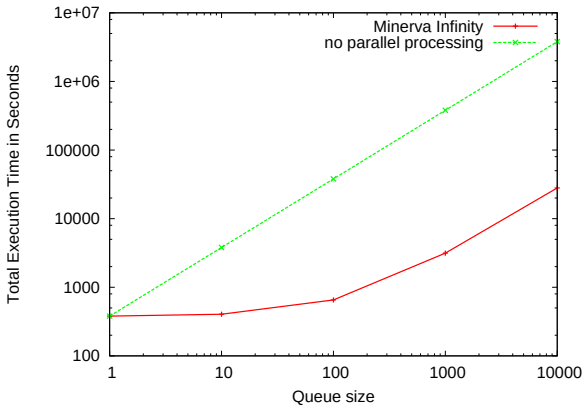

Figure 8.11: Scalability Results

Chapter 9

Conclusion and Outlook

We have considered distributed top-k algorithms, where the index lists for the query attributes are spread across multiple peers.

We have presented the KLEE framework for distributed top-k query processing. KLEE's salient features set it apart from related work in several ways. First, KLEE makes for the first time a strong case for approximate top-k algorithms in widely distributed environments. Second, KLEE allows the trading-off of result quality vs performance. KLEE even allows for trading-off between bandwidth vs the number of communication phases. Experiments show that KLEE achieves great performance gains in network bandwidth, query response times, and local peer load, and high quality results. Moreover, we have presented means to model score distributions and how these score models can be used to reason about parameter values that play an important role in the overall performance of KLEE. We have presented the family of GRASS algorithms that use these score models for optimizing the query execution. GRASS shows significant performance gains, in particular for queries that involve many data sources.

We have derived probabilistic guarantees for the algorithms presented in this thesis, to show both analytically and by a comprehensive experimental study their suitability for various application classes.

Furthermore, we have presented Minerva∞, a novel architecture for a peer-to-peer web search engine. The key distinguishing feature of Minerva∞ is its high-levels of distribution for both data and processing.

Meaningful cost predictions in distributed systems mainly depends on an accurate model that reflects the network characteristics as well as the processing power of the involved peers. In case of dramatic changes of the peers' available network bandwidth or processing power during a query's run-time, one could even think about dynamically reorganizing the whole query execution plan. This is left for future work.

The query processing algorithm in Minerva∞ incorporates the principles of the NRA algorithm, properly adapted to the networked infrastructure. Al-

ternatively, we could think about incorporating different top-k algorithms. In particular, TPUT can be used to reduce the number of phases to three, at the cost of incurring a possibly large number of random accesses during TIN processing. Alternatively, KLEE or GRASS could be used to ensure a small number of phases (two or three) with significant further savings in response times and network bandwidth, but with approximate answers. Another interesting idea for future work is the dynamic adaption of the TIN sizes in Minerva∞ and the data replication strategies to changing workloads.

Appendix A

Appendix

A.1 Benchmark Queries

mining gold silver coal	juvenile delinquency
Lewis and Clark expedition	wireless communications
pest control safety	physical therapists
cotton industry	computer viruses
genealogy searches	Physical Fitness
folk art folk music	legalization of marijuana
Schizophrenia	Agricultural biotechnology
cell phones	Emergency and disaster preparedness assistance
Polygraphs	Shipwrecks
Cybercrime, internet fraud, and cyber fraud	children's literature
cartography	Veteran's Benefits
Photography	Air Bag Safety
death penalty	Nuclear power plants
affirmative action	Early Childhood Education
Asbestos	Counterfeit money
deafness in children	wildlife conservation
food safety	Literacy
arctic exploration	global warming
coin collecting	weather hazards and extremes
National Public Radio/TV	North Korea
Electric Automobiles	homelessness
forest fires	Ozone layer
Bicycle trails	infant mortality
trains/railroads	robots
Bilingual education	anthrax

Table A.1: 50 .GOV Queries

genre:Western actor:Wayne_John actor:Hepburn_Katherine Sheriff Marshall
genre:Western actor:Fonda_Henry Outlaw
genre:Western actor:Newman_Paul Outlaw
genre:Western actor:Wayne_John Indians
genre:Action actor:Reeves_Keanu Martial Arts Fight
genre:Thriller actor:Pitt_Brad actor:Freeman_Morgan Murder
genre:Thriller actor:Schwarzenegger_Arnold Robot
genre:Comedy actor:Allen_Woody Woman
genre:Comedy Tom Hanks Vietnam War
genre:SciFi actor:Roberts_Julia Alien Space
genre:SciFi actor:Ford_Harrison Space War Battle
genre:Film-Noir genre:Thriller actor:Marlowe_Frank Chicago Prohibition
genre:Drama actor:Ozari_Romano Nosferatu
genre:Drama actor:Seymour_Dan World War
genre:Thriller actor:Bogart_Humphrey Casablanca
actor:Welles_Orson Rosebud
genre:Thriller 3rd Man
genre:Horror actor:Lee_Christopher Coffin Blood Vampire
genre:Crime actor:Sims_Joan Marple Paddington
genre:Action actor:Dalton_Dimothy SPECTRE

Table A.2: 20 IMDB Queries

mining gold silver coal metal location mineral resources industry
juvenile delinquency youth minor crime law jurisdiction offense prevention
Lewis and Clark expedition historic explore
wireless communications radio broadcasting transmission electromagnetic waves use research technology regulations legislative
pest control safety epidemic contamination quarantine
physical therapists healer training licensing skills body
cotton industry growing harvesting cloth silky fiber plant fabric textile material
computer viruses software program malevolent worm trojan bug
genealogy searches family tree lineage bloodline descent ancestry pedigree origin parentage generation
Physical Fitness shape condition body training
folk art folk music ethnic traditional song ballad country western gospel singing
legalization marijuana cannabis drug soft leaves plant smoked chewed euphoric abuse substance possession control pot grass dope weed smoke
Schizophrenia disorder psychosis distortion reality disturbance social contact
Agricultural biotechnology farming cultivation land food grow crops microorganism bacteria industrial process genetically altered
cell phones cellular mobile hand-held radio transmitter receiver wireless telephone electronic signal sound
Emergency disaster preparedness assistance local state national crisis danger immediate action catastrophe extreme readiness help aid
Polygraphs requirement exam medical instrument physiological process pulse rate blood pressure respiration perspiration lie detector
Shipwrecks ship wreck accident sea capsizing boat nautical water
Cybercrime internet fraud cyber detection crime
children's literature youngster kid book writing novel
cartography mapmaking map chart
Veteran's Benefits ex-serviceman financial assistance
Photography picture taking telephotography
Air Bag Safety restraint automobile inflate collision
death penalty execution executing capital punishment hanging electrocution decapitation beheading crucifixion burning

Table A.3: 25 Extended GOV (XGOV) Queries. Part 1

Nuclear atomic power plants power station power house
affirmative action discrimination minority groups
Early Childhood child infancy babyhood Education instruction teaching pedagogy elementary
Asbestos fibrous amphibole asbestosis
Counterfeit imitation forgery fake false forged money paper coin
deafness deaf hearing loss deaf-mutism deaf-muteness in children child kids youngsters preschooler infant baby
wildlife living undomesticated conservation preservation conservancy environment
food nutrient foodstuff comestible edible eatable eat safety risklessness security
Literacy center ability read write human skills learn knowledge cognition
arctic north-polar north pole exploration geographical expedition discovery
global warming increase average temperature earth atmosphere climatic changes planetary worldwide heating
coin collecting numismatics numismatology coin collection
weather hazards and extremes peril risk jeopardy wind rain snow storm wave
National Public Radio/TV television telecasting broadcasting cable
North Korea Democratic People's Republic of Korea DPRK communist country
Electric Automobiles production car research progress fuel
homelessness combat vagrancy wandering livelihood home prevalence
forest fires woods burn flames dry summer
Ozone layer environment pollution ultraviolet rays industry
Bicycle trails mountain bike downhill sport offroad nature
infant mortality deathrate children neonatal
trains/railroads travel safety government industry
robots artificial machine production lane research
Bilingual education language learning skills school children
anthrax bacillus anthracis fever disease treatment prevention contagion quarantine

Table A.4: 25 Extended GOV (XGOV) Queries. Part 2

List of Figures

List of Algorithms

List of Tables

Bibliography

[Abe01] Karl Aberer. P-grid: A self-organizing access structure for p2p information systems. In Carlo Batini, Fausto Giunchiglia, Paolo Giorgini, and Massimo Mecella, editors, *Cooperative Information Systems, 9th International Conference, CoopIS 2001, Trento, Italy, September 5-7, 2001, Proceedings*, volume 2172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2001.

[ACDG03] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, and Aristides Gionis. Automated ranking of database query results. In *1st Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, January 5-8, 2003*, 2003.

[ACKS06] Rakesh Agrawal, Alvin Cheung, Karin Kailing, and Stefan Schönauer. Towards traceability across sovereign, distributed rfid databases. In Parisa Ghodous, Rose Dieng-Kuntz, and Geilson Loureiro, editors, *Leading the Web in Concurrent Engineering. Next Generation Concurrent Engineering, Proceedings of the 13th ISPE International Conference on Concurrent Engineering (ISPE CE 2006), September 18-22, 2006, Antibes, France.*, pages 174–184. IOS Press, 2006.

[ADH05] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. P-grid: Dynamics of self-organizing processes in structured peer-to-peer systems. In Ralf Steinmetz and Klaus Wehrle, editors, *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 2005.

[AdKM01] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In W. Bruce Croft, David J. Harper, Donald H. Kraft, and Justin Zobel, editors, *SIGIR 2001: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, September 9-13, 2001, New Orleans, Louisiana, USA*, pages 35–42. ACM, 2001.

[All90] Arnold O. Allen. *Probability, statistics, and queueing theory with computer science applications*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[APV06] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Reducing network traffic in unstructured p2p systems using top- queries. *Distributed and Parallel Databases*, 19(2-3):67–86, 2006.

[AS03] James Aspnes and Gauri Shah. Skip graphs. *The Computing Research Repository (CoRR)*, cs.DS/0306043, 2003.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[BCG02] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[Ben07] Matthias Bender. *Advanced Methods for Query Routing in Peer-to-Peer Information Retrieval*. PhD thesis, Universität des Saarlandes, 2007.

[BGM02] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over web-accessible databases. In *Proceedings of the 18th International Conference on Data Engineering, ICDE, 26 February - 1 March 2002, San Jose, CA*, pages 369–. IEEE Computer Society, 2002.

[BGRS99] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In Catriel Beeri and Peter Buneman, editors, *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings.*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer, 1999.

[BJRS03] Mayank Bawa, Roberto J. Bayardo Jr., Sridhar Rajagopalan, and Eugene J. Shekita. Make it fresh, make it quick: searching a network of personal webservers. In *WWW*, pages 577–586, 2003.

[BKK+01] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Donald Kossmann, Stefan Seltzsam, and Konrad Stocker. Objectglobe: Open distributed query processing services on the internet. *IEEE Data Eng. Bull.*, 24(1):64–70, 2001.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BM05] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[BMT+05a] Matthias Bender, Sebastian Michel, Peter Triantafillou, Gerhard Weikum, and Christian Zimmer. Improving collection selection with overlap awareness in p2p search engines. In Ricardo A. Baeza-Yates, Nivio Ziviani, Gary Marchionini, Alistair Moffat, and John Tait, editors, *SIGIR 2005: Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Salvador, Brazil, August 15-19, 2005*, pages 67–74. ACM, 2005.

[BMT+05b] Matthias Bender, Sebastian Michel, Peter Triantafillou, Gerhard Weikum, and Christian Zimmer. Minerva: Collaborative p2p search. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1263–1266. ACM, 2005.

[BMTW06] Matthias Bender, Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Global document frequency estimation in peer-to-peer web search. In Dayou Zhou, editor, *9th International Workshop on the Web and Databases (WebDB 2006)*, pages 69–74, Chicago, USA, 2006.

[BNST05] Wolf-Tilo Balke, Wolfgang Nejdl, Wolf Siberski, and Uwe Thaden. Progressive distributed top k retrieval in peer-to-peer networks. In *Proceedings of the 21st International Conference on Data Engineering, ICDE, 5-8 April 2005, Tokyo, Japan*, pages 174–185. IEEE Computer Society, 2005.

[BO03] Brian Babcock and Chris Olston. Distributed top-k monitoring. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 28–39. ACM, 2003.

[BSVW99] Tom Brijs, Gilbert Swinnen, Koen Vanhoof, and Geert Wets. Using association rules for product assortment decisions: A case study. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 15-18, 1999, San Diego, CA, USA.*, pages 254–260. ACM, 1999.

[Cal00] J. Callan. *Distributed Information Retrieval*, chapter In B. Croft, editor, Advances in Information Retrieval: Recent Research from the Center for Intelligent Information Retrieval. The Kluwer International Series on Information Retrieval., pages 127–150. 2000.

[CAPMN03] Francisco Matias Cuenca-Acuna, Christopher Peery, Richard P. Martin, and Thu D. Nguyen. Planetp: Using gossiping to build content addressable peer-to-peer information sharing communities. In *12th International Symposium on High-Performance Distributed Computing (HPDC-12 2003), 22-24 June 2003, Seattle, WA, USA*, pages 236–249. IEEE Computer Society, 2003.

[CDHW04] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic ranking of database query results. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 888–899. Morgan Kaufmann, 2004.

[CG95] K. Church and W. Gale. Poisson mixtures. *Natural Language Engineering*, 1(2):163–190, 1995.

[CG96] Surajit Chaudhuri and Luis Gravano. Optimizing queries over multimedia repositories. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 91–102. ACM Press, 1996.

[CGM04] Surajit Chaudhuri, Luis Gravano, and Amélie Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Trans. Knowl. Data Eng.*, 16(8):992–1009, 2004.

[Cha02] Soumen Chakrabarti. *Mining the Web: Discovering Knowledge from HyperText Data*. Science & Technology Books, 2002.

[CL03] W. Bruce Croft and John Lafferty. *Language Modeling for Information Retrieval*, volume 13. Kluwer International Series on Information Retrieval, 2003.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 2001.

[CP02] Paolo Ciaccia and Marco Patella. Searching in metric spaces with user-defined and approximate distances. *ACM Trans. Database Syst.*, 27(4):398–437, 2002.

[CW04] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 206–215. ACM, 2004.

[CwH02] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 346–357. ACM, 2002.

[DEB05] Special issue on in-network query processing. *IEEE Data Eng. Bull.*, 28(1), 2005.

[DKM+06] Micah Dubinko, Ravi Kumar, Joseph Magnani, Jasmine Novak, Prabhakar Raghavan, and Andrew Tomkins. Visualizing tags over time. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, pages 193–202. ACM, 2006.

[DN03] Herbert A. David and Haikady N. Nagaraja. *Order Statistics*. John Wiley & Sons, 3rd edition, August 2003.

[DR01] Peter Druschel and Antony I. T. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems, May 20-23, 2001, Elmau/Oberbayern, Germany*, pages 75–80. IEEE Computer Society, 2001.

[dVMNK02] Arjen P. de Vries, Nikos Mamoulis, Niels Nes, and Martin L. Kersten. Efficient k-nn search on vertically decomposed data. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 322–333. ACM, 2002.

[Fag99] Ronald Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.

[Fag02] Ronald Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.

[FCAB98] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 31 - September 4, 1998, Vancouver, B.C., Canada.*, pages 254–265. ACM, 1998.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[GBK00] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 419–428. Morgan Kaufmann, 2000.

[GBK01] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628. IEEE Computer Society, 2001.

[GF98] David A. Grossman and Ophir Frieder. *Information Retrieval: Algorithms and Heuristics*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[Goo95] Michael T. Goodrich. Efficient piecewise-linear function approximation using the uniform metric. *Discrete & Computational Geometry*, 14(4):445–462, 1995.

[Har75] S. Harter. A probabilistic approach to automatic keyword indexing (part 1). *Journal of the American Society for Computer Science*, 24(4):197–206, 1975.

[HCH+05] Ryan Huebsch, Brent N. Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The architecture of pier: an internet-scale query processor. In *2nd Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, January 4-7, 2005*, pages 28–43, 2005.

[HD05] Emir Halepovic and Ralph Deters. The jxta performance model and evaluation. *Future Generation Comp. Syst.*, 21(3):377–390, 2005.

[HJS+03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[HS03] Gísli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Y. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, pages 190–201, 2000.

[KCR06] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 289–300. ACM, 2006.

[KKNR04] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 779–790. ACM, 2004.

[KLL+97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, STOC, El Paso, Texas, USA, May 4-6, 1997.*, pages 654–663. ACM, 1997. ISBN 0-89791-888-6, 1997.

[KOT04] Nick Koudas, Beng Chin Ooi, Kian-Lee Tan, and Rui Zhang 0003. Approximate nn queries on streams with guaranteed error/performance bounds. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 804–815. Morgan Kaufmann, 2004.

[LC03] Jie Lu and James P. Callan. Content-based retrieval in hybrid peer-to-peer networks. In *Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management, New Orleans, Louisiana, USA, November 2-8, 2003*, pages 199–206. ACM, 2003.

[LKP+06] Toan Luu, Fabius Klemm, Ivana Podnar, Martin Rajman, and Karl Aberer. Alvis peers: a scalable full-text peer-to-peer retrieval engine. In *P2PIR '06: Proceedings of the international workshop on Information retrieval in peer-to-peer networks*, pages 41–48, New York, NY, USA, 2006. ACM Press.

[LNS96] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. Lh* - a scalable, distributed data structure. *ACM Trans. Database Syst.*, 21(4):480–525, 1996.

[LS03] Xiaohui Long and Torsten Suel. Optimized query execution in large search engines with global page ordering. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 129–140. Morgan Kaufmann, 2003.

[MBG04] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[MBN+06] Sebastian Michel, Matthias Bender, Nikos Ntarmos, Peter Triantafillou, Gerhard Weikum, and Christian Zimmer. Discovering and exploiting keyword and attribute-value co-occurrences to improve p2p routing indices. In Philip S. Yu, Vassilis J. Tsotras, Edward A. Fox, and Bing Liu, editors, *CIKM*, pages 172–181. ACM, 2006.

[MBTW06] Sebastian Michel, Matthias Bender, Peter Triantafillou, and Gerhard Weikum. Iqn routing: Integrating quality and novelty in p2p querying and ranking. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 2006.

[MEH05] Wolfgang Müller, Martin Eisenhardt, and Andreas Henrich. Scalable summary based retrieval in p2p networks. In Otthein Herzog, Hans-Jörg Schek, Norbert Fuhr, Abdur Chowdhury, and Wilfried Teiken, editors, *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pages 586–593. ACM, 2005.

[MFHH05] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[MSDO05] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. Finding (recently) frequent items in distributed data streams. In *Proceedings of the 21st International*

Conference on Data Engineering, ICDE, 5-8 April 2005, Tokyo, Japan, pages 767–778. IEEE Computer Society, 2005.

[MTW05a] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Klee: A framework for distributed top-k query algorithms. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 637–648. ACM, 2005.

[MTW05b] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. $\text{Minerva}_{\text{infinity}}$: A scalable efficient peer-to-peer search engine. In Gustavo Alonso, editor, *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 60–81. Springer, 2005.

[NCS+01] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 281–290. Morgan Kaufmann, 2001.

[NM07] Thomas Neumann and Sebastian Michel. Algebraic query optimization for distributed top-k queries. In Alfons Kemper, editor, *Datenbanksysteme in Business, Technologie und Web, 12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Aachen, Germany, 7.-9. März 2007*, LNI. GI, 2007.

[NR99] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 22–29. IEEE Computer Society, 1999.

[PAP+03] Evaggelia Pitoura, Serge Abiteboul, Dieter Pfoser, George Samaras, and Michalis Vazirgiannis. Dbglobe: a service-oriented p2p system for global computing. *SIGMOD Record*, 32(3):77–82, 2003.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical recipes in C: the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1988.

[PMW07] Josiane Xavier Parreira, Sebastian Michel, and Gerhard Weikum. p2pdating: Real life inspired semantic overlay networks for web search. *Inf. Process. Manage.*, 43(3):643–664, 2007.

[Pow98] M. J. D. Powell. A "taut string algorithm" for straightening a piecewise linear path in two dimensions. *IMA J. Numer. Anal.*, 18(1), Jan 1998.

[PZSD96] Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *JASIS*, 47(10):749–764, 1996.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, San Diego, CA, USA.*, pages 161–172. ACM, 2001.

[RV03] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In Markus Endler and Douglas C. Schmidt, editors, *Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings*, volume 2672 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2003.

[SCC+01] Aya Soffer, David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, and Yoëlle S. Maarek. Static index pruning for information retrieval systems. In W. Bruce Croft, David J. Harper, Donald H. Kraft, and Justin Zobel, editors, *SIGIR 2001: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, September 9-13, 2001, New Orleans, Louisiana, USA*, pages 43–50. ACM, 2001.

[SL00] D. Salomoni and S. Luitz. High performance throughput tuning/measurement. http://www.slac.stanford.edu/grp/scs/net/talk/High_perf_ppdg_jul2000.ppt. 2000.

[SMK+01] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, San Diego, CA, USA.*, pages 149–160. ACM, 2001.

[SMwW+03] Torsten Suel, Chandan Mathur, Jo wen Wu, Jiangong Zhang, Alex Delis, Mehdi Kharrazi, Xiaohui Long, and Kulesh Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In Vassilis Christophides and Juliana Freire, editors, *WebDB*, pages 67–72, 2003.

[SSK06] Izchak Sharfman, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *A geometric approach to monitoring threshold functions over distributed data streams.*, pages 301–312. ACM, 2006.

[Tir03] Ajay Tirumala et al. iperf: Testing the limits of your network. http://dast.nlanr.net/projects/iperf/. 2003.

[TP03] Peter Triantafillou and Theoni Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In Karl Aberer, Vana Kalogeraki, and Manolis Koubarakis, editors, *Databases, Information Systems, and Peer-to-Peer Computing, First International Workshop, DBISP2P, Berlin Germany, September 7-8, 2003, Revised Papers*, volume 2944 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2003.

[TWS04] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 648–659. Morgan Kaufmann, 2004.

[VBW98] Radek Vingralek, Yuri Breitbart, and Gerhard Weikum. Snowball: Scalable storage on networks of workstations with balanced load. *Distributed and Parallel Databases*, 6(2):117–156, 1998.

[Was04] Larry Wasserman. *All of Statistics*. Springer, 2004.

[WGD03] Yuan Wang, Leonidas Galanis, and David J. DeWitt. Galanx: An efficient peer-to-peer search engine system. Technical report, University of Wisconsin - Madison,, 2003.

[YLW+05] Hailing Yu, Hua-Gang Li, Ping Wu, Divyakant Agrawal, and Amr El Abbadi. Efficient processing of distributed top- queries. In Kim Viborg Andersen, John K. Debenham, and Roland Wagner, editors, *Database and Expert Systems Applications, 16th International Conference, DEXA 2005, Copenhagen, Denmark, August*

22-26, 2005, Proceedings, volume 3588 of *Lecture Notes in Computer Science*, pages 65–74. Springer, 2005.

[YPM03] Clement T. Yu, George Philip, and Weiyi Meng. Distributed top-n query processing with possibly uncooperative local systems. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 117–128. Morgan Kaufmann, 2003.

[YSMQ01] Clement T. Yu, Prasoon Sharma, Weiyi Meng, and Yan Qin. Database selection for processing k nearest neighbors queries in distributed environments. In *Proceedings of ACM/IEEE Joint Conference on Digital Libraries, JCDL 2001, Roanoke, Virginia, USA, June 24-28, 2001*, pages 215–222. ACM, 2001.

[Zip49] George Kingsley Zipf. *Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology.* Addison-Wesley, 1949.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, 2001.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[ZOWX06] Qi Zhao, Mitsunori Ogihara, Haixun Wang, and Jun Xu. Finding global icebergs over distributed data sets. In Stijn Vansummeren, editor, *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, Maryland, USA*, pages 298–307. ACM, 2006.

[ZYVG+05] D. Zeinalipour-Yazti, Z. Vagena, D. Gunopulos, V. Kalogeraki, V. Tsotras, M. Vlachos, N. Koudas, and D. Srivastava. The threshold join algorithm for top-k queries in distributed sensor networks. In *2nd International VLDB Workshop on Data Management for Sensor Networks.*, 2005.

Index