KLEE: A Framework for Distributed Top-k Query Algorithms

Sebastian Michel
Max-Planck Institute for Informatics
Saarbrücken, Germany
smichel@mpi-inf.mpg.de

Peter Triantafillou
University of Patras
Rio, Greece
peter@ceid.upatras.gr

Gerhard Weikum
Max-Planck Institute for Informatics
Saarbrücken, Germany
weikum@mpi-inf.mpg.de

Abstract

This paper addresses the efficient processing of top-k queries in wide-area distributed data repositories where the index lists for the attribute values (or text terms) of a query are distributed across a number of data peers and the computational costs include network latency, bandwidth consumption, and local peer work. 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. Our evaluation employed real-world and synthetic large, web-data collections, and query benchmarks. Our experimental results show that KLEE can achieve major performance gains in terms of network bandwidth, query response times, and much lighter peer loads, all with small errors in result precision and other result-quality measures.

1. Introduction

1.1 Motivation

Top-k query processing has received much attention in a variety of settings such as similarity search on multimedia data [CGM04, NR99, Fa99, GKB00, Bey99, Na01, deV02], ranked retrieval on text and semi-structured documents in digital libraries and on the Web [AKM01, LS03, TWS04, Kau04, Ba03, So01, PZS96, Yu01], spatial data analysis [BBK01, CP02, HS03], network and stream monitoring [BO03, Kou04, CW04] collaborative recommendation and preference queries on e-commerce product catalogs [YPM03, MGB04, BGM02, GKB01, CH02], and ranking of SQL-style query results on structured data sources in general [Ag03, Ch04, BCG02]. In terms of efficiency, the most successful approaches are based on the family of threshold algorithms (TA) originally developed by [FLN03, GKB00, 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 [Hue05] 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.

In this paper we assume that index lists for text terms or data attributes are distributed across peers. Index lists are crucial for a TA-style top-k algorithm; in their distributed processing we are judicious about the resulting communication costs: a) network latency incurred by message rounds and b) network bandwidth consumption incurred by the data exchange among the peers that collaborate on behalf of a given query. Moreover, to limit the local processing costs of each peer, we consider TA-sorted variants (aka. NRA) [FLN03, GKB01] that disallow random accesses to index list entries and rather limit themselves to sorted accesses, which are 20 times faster for large, disk-resident index lists. We also consider applying probabilistic approximations to the true top-k result, using techniques like the ones in [TWS04]. Such relaxations are well justified for most applications of top-k queries, where both the underlying score functions and the result interpretation by the user have a heuristic nature anyway. In wide-area P2P systems, approximation techniques are even more appropriate as we face significant tradeoffs between execution cost and search result quality.

1.2 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

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

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).

In the following we use only IR-style terminology, speaking of "terms" and "documents", for simplicity. Each peer P_j stores one index list, $I_j(t)$, over a term t. $I_j(t)$ consists of a number of *(docID, score)* pairs, where *score* is a real number in (0, 1] reflecting the significance of the document with *docID* for term *t*. Each index list is assumed to be sorted in descending order according to *score*. In general, *score(docID)* reflects the score associated with *docID* in an index list, e.g., a *tf*idf*-style (term_frequency*inverse_document_frequency) or language-model-based measure derived from term frequency statistics.

A query, $q(T, k)$, initiated at a peer P_{init}, consists of a nonempty set of terms, $T = \{t_1, t_2, \ldots t_t\}$, and an integer k. Assuming the existence of a set of, say m, peers having the most relevant index lists for the terms in T, with $m \leq t$, our task is to devise efficient methods for P_{init} to access these distributed index lists at the m peers, so as to produce the list of (the IDs of) the top-k documents for the term set T. The top-k result is the sorted list in descending order of *TotalScore* which consists of pairs *(docID, TotalScore)*, where *TotalScore* for a document with ID *docID* is a monotonic aggregation of the scores of this document in all m index lists. For the sake of concreteness, we will use summation for score aggregation, but weighted sums and other monotonic functions are supported, too. In case an index list does not contain a particular *docID*, its score for *docID* is set to zero, when calculating its *TotalScore*. Note that P_{init} serves as a coordinator only for the given query; different queries are usually coordinated by different peers.

A naïve solution would be to have all m cohort peers send the complete index lists to P_{init} and then execute a centralized TA-style method on the copied lists at P_{init}. This approach is unacceptable in a P2P system for its waste of network bandwidth resulting from transferring complete index lists.

An alternative approach would be to execute TA at P_{init} and access the remote index lists one entry at a time as needed. This method is equally undesirable for it incurs many small messages and needs a number of message rounds that is equal to the maximum index-scan depth among the participating peers. Even when messages are batched (e.g., with 100 successive index entries in a single message), the total latency of many message rounds renders this approach unattractive.

As recently shown by [CW04], a good network-cost-conscious algorithm should ensure that distributed top-k algorithms terminate within a fixed, small number of phases. In each phase P_{init}, acting as a coordinator, receives information from the peers' index lists and then tries to intelligently estimate whether it has enough information to compute a high quality approximation of the top-k list and stop the process as early as possible. The number of phases should ideally be constant, to guarantee acceptable latency, while requiring the cohort peers to send only as little as possible information, aiming to minimize network bandwidth consumption. To this end the coordinator needs to address two major issues:

1) *Missing scores* are an issue when document IDs are included in the responses of some peers and not in the responses of others. This complicates the estimation of the total scores for these documents, which in turn makes it difficult to prune top-k candidates early or at least to produce a high-quality top-k approximation.

2*) Missing documents* are an issue in the coordinator's incomplete view of the documents that are candidates for the top-k result. It is difficult for the coordinator to learn about documents with relatively low scores (and deeper positions) in the index lists of all peers; such documents may nevertheless be good candidates for the top-k result if their total score, summed up over all index lists, is high.

1.3 Contributions

This paper presents a novel family of algorithms for distributed top-k query processing, coined KLEE. The name of the algorithm refers to the plant known as clover in English: KLEE uses three or, optionally for additional optimization, four algorithmic steps.

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-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.

2. Related Work

Among the ample work on top-k query processing (see the references in Section 1.1), the TA family of algorithms for monotonic score aggregation [FLN03, GBK00, NR99] stands out as an extremely efficient and highly versatile method. The current paper builds on the TA-sorted (aka. NRA) variant which processes 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 i^{th} 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) < *topKscore* can be safely dismissed, where *topKscore* 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 = $\Sigma_{i=1..m}$ high(i) can not qualify for the top-k either).

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 topKscore value drops below some threshold: $P[worstscore(d) + \Sigma_i S(i) > topKscore] < \varepsilon$, 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.

The first distributed TA-style algorithm has been presented in [BGM02, MGB04]. 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 [CH02]. In our widely distributed setting, none of these scheduling methods are relevant for they still incur an unbounded number of message rounds.

The method in [Su03] 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.

The recent work by [Ba05] addresses the optimization of communication costs in P2P networks. However, the emphasis is on appropriate topologies for overlay networks. The paper develops efficient routing methods among super-peers in a hypercube topology.

The TPUT algorithm by [CW04] is closest to our KLEE approach; their architectural goal of a fixed, small number of communication phases has also influenced KLEE, but our design philosophy is broader in a number of dimensions. TPUT executes TA in three phases: 1) fetch the k best (DocID, Score) entries from each cohort peer and compute the *topKscore* using zero-score values for all missing scores; 2) ask each of the *m* cohort peers for entries with *Score* > *topKscore* / *m*, then compute a better topKscore value and eliminate candidates whose bestscore is not higher than topKscore; 3) fetch the still missing scores for the remaining candidates, asking the cohorts to do random accesses.

3. 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.

3.1 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 $S = \{\alpha_1, \alpha_2, \ldots, \alpha_s\}$. 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 $PFP \approx (1 - e^{-hs/b})^h$ [Bl70, Fan98], 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*.

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.

3.2. 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, *topKscore*. 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 *topKscore/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.

3.3 The Candidate List Filters Matrix Structure

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 *topKscore* 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 *topKscore / 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 *topKscore*) 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, that is find those docIDs associated locally with a score that is better than (*topKscore / m*) and
2. create a bitmap filter of this set, called the peer's *Candidate List Filter*, CLF. Specifically, for each docID with *score(docID) > (topKscore / m)*, the peer will hash the docID and set the proper bit in its *CLF*.

P_{init} utilizing the histogram statistics received, can know from the 1st phase the number of documents at each peer that have a better score than *topKscore / 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' *CLFs* will have the same size, *b*. When P_{init} receives these *CLFs* it constructs a bitmap matrix, the *CLF Matrix*. *The CLF Matrix*:

- is an m × b matrix,
- its i-th row is the CLF received from the *i*-th peer.

3.4 Harvesting Candidate List Filters

The rationale for building the *CLF Matrix* is that, by construction, all docIDs (from all *m* peers) which have a higher score than the *topKscore/m* in R of the *m* peers, will be hashed into a *column* of the *CLF 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 *topKscore/m* in only a small number of peers, making the likelihood of these docIDs having a total score better than *topKscore* 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 *CLF 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 *CLF 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 List Filters before hand, so to avoid having an additional communication phase if they are not needed.

4. The KLEE Algorithmic Framework

4.1 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 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*. P_{init} communicates with the *m* cohort peers in order to produce a good estimation of the *topKscore*, 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) > (topKscore / 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 list filters that would be constructed by the cohort peers.
3. *The Candidate List 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 Candidate List Filter Matrix data structure. Using the latter, a new set of per-peer candidate lists are constructed, replacing the ones constructed in the first step. Specifically, for a peer P_i its candidate list is defined to contain those docIDs for which *hash(docID)* is one of the columns of the *CLF Matrix* with enough bits set.
4. *The Candidate List 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.

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

$$\{docID : docID \in I_j \wedge score(docID) > topKscore / m\}$$

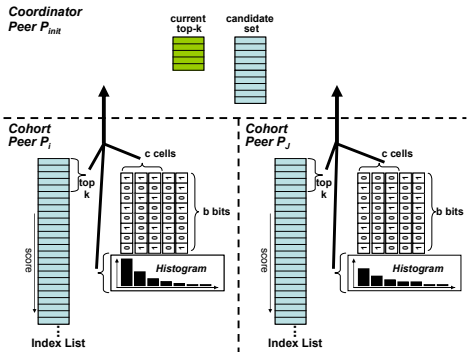


Figure 1: Two peers responding to P_{init}

4.4 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 *topKscore/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 CLF construction, peers use just one hash function.

Arguing about the expected values of the *CLF Matrix*, we note that the probability of any bit of a column being set (independently by a peer in its *CLF filter*) is given by $P_1 = 1/lf$ where, *lf* is the load factor for the Bloom filter which is given by: $lf = b/d$ where b is the size of the peers' *CLFs*. Next, the key value to estimate is the expected number of columns of the *CLF 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, C_r, 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 *CLF* 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 *CLF* filter, 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 $C_{r,3} = P_R \times C_0$ making the value of P_R the key to the expected savings in the bandwidth in the last phase of the algorithm.

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.

4.5 The Candidate List Reduction Step.

The following details step 3 of KLEE, which revolves around the construction and manipulation of the peers' *CLF* structures.

Candidate List 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 topKscore / 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 CLF: using just one hash function and a bitmap with size b = *load_factor* × *max_size_candidate_list*, with a load_factor value large enough to ensure low probabilities of false positives. The *CLF* 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 *CLF* bit matrix, *CLFM*, of size $m \times b$. As mentioned, the rows in this matrix are the *CLF* filters received from the peers: *CLFM [i,j]* represents

the jth entry in peer P_i's *CLF* filter for *candidate_docs*(P_i).

4. P_{init} defines the *interesting columns* of its *CLFM* 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 *CLF*.

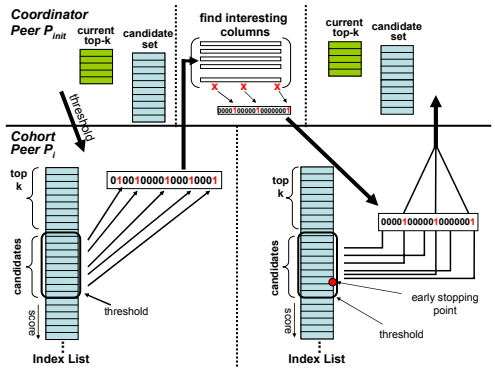


Figure 2: Constructing CLFM from CLFs

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

4.6 The Candidate List 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.

5. 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 and (ii) the number of bits, *R*, that have to be set in order for any column of the *CLFM* 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 *CLFs* 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 *CLFM* 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 CLFM, 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 paper. 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 *CLFs* that are no longer simple bit maps: a non-zero value in a *CLF* 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 CLF position indicated by the hash, the cell number of the peer's histogram into which this docID belongs. Formally, CLF[i] = r, if and only if *hash(docID) = i*, and $lb[r] \leq score(docID) \leq ub[r]$.
2. P_{init} after receiving the peer *CLFs* constructs as before the $m \times b$ matrix *CLFM*.
3. Finally, P_{init} defines a column of *CLFM*, $j, 1 \leq j \leq b$, as interesting if and only if:

$$\sum_{i=1}^{m} ub_i[CLFM[i, j]] > topKscore$$

where $ub_i[r]$ 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 *CLFM* columns ensures that no docID that could attain a *TotalScore* higher than *topKscore* would be missed.

Obviously, the new definition of the interesting columns of the *CLFM* 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 *CLFs*. Since cell numbers are stored now in *CLFs*, 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.

6. Experimentation

6.1 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* * *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, 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 (http://www.cogsci.princeton.edu/~wn/). 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.

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 [Zi49], 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 θ = 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 Ω. 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, Ω] and this docID was inserted in $I(t_2)$ at position *v*. By controlling the value of Ω between [k+1, sizeof($I(t_2)$)], we create stronger or weaker correlations (for smaller or greater values of Ω, 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.

6.2 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 CLFs 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.

6.3 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 *topKscore*. We also computed the footrule distance for the ranks of approximate vs. exact top-k results.

6.4 Experiments

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 (5%, 10%, and 20%) of the total score mass. For space reasons, we show only the results with $c = 10\%$ of the score mass. For the value of R, we used the technique of Section 5 to select the interesting columns.

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 *topKscore* due to false positives. For the 3nd step, the size of peers' CLFs 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 CLFs 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 [SaLu00]. 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 [SaLu00] using NLANR's iPerf tool [TQDFG03].

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.

6.5 Performance Results

6.5.1 On Synthetic Benchmarks

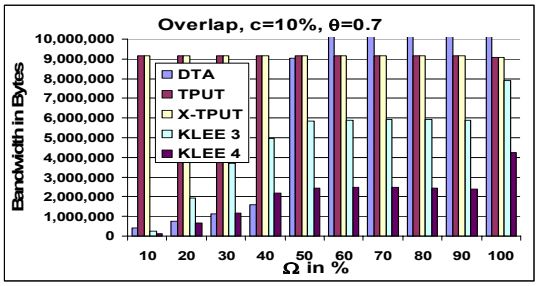
Bandwidth Costs. Figure 3 shows the bandwidth results for Overlap. We show results for $\theta = 0.7$, and t=5-term queries, (similar results occur with all other tested values of θ and t, and are omitted for space reasons). Ω was varied to correspond to the index list positions capturing from 10% to 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 topKscore, 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 CLFM 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 and 5 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.



Figures 3: Bandwidth for Overlap

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 1, 2, and 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 1, 2, and 3, which show total benchmark times (i.e., for the entire batch of 50 queries). In Table 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.

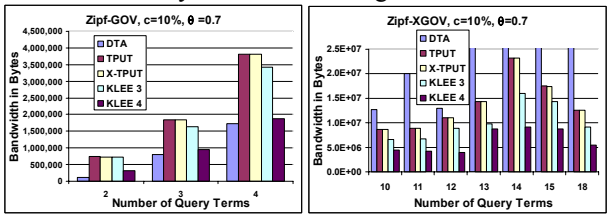


Figure 4,5: Bandwidth for Zipf-GOV and Zipf-XGOV

Result Quality. Tables 1, 2 and 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.

6.5.2 On Real-World Benchmarks

Bandwidth Costs. Figures 6 and 8 and the first columns of Tables 4, 5 and 6 show the bandwidth results for GOV, XGOV, and IMDB respectively. Figure 7 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.

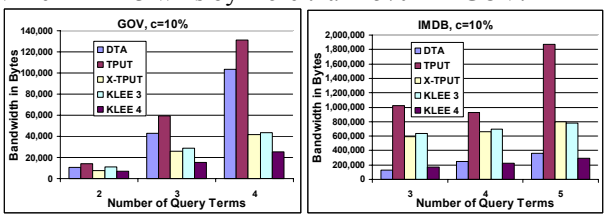


Figure 6, 7: Bandwidth for GOV and IMDB

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

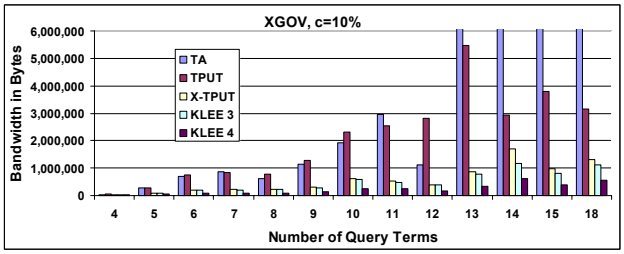


Figure 8: Bandwidth for XGOV

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 6, 7, and 8 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

Table 1: Aggregated Statistics for the Overlap Benchmark with θ = 0.7, Ω=30% and c=10%

Overlap+Zipf c=10%,θ= 0.7,Ω=30%	Total # of Bytes	Total Time in ms	Average Recall	Avg Score Error/topKScore	Avg Rank Distance	# Sorted Accesses	# Random Accesses
DTA	1,146.320	157,420	1	0	0	8,060	150
TPUT	9,150,904	29,270	1	0	0	70,867	0
X-TPUT	9,150,904	28,335	1	0	0	70,867	0
KLEE 3	3,678,780	12,971	0.92	0.0003	1.45	27,801	0
KLEE 4	1,192,704	6,546	0.91	0.0003	1.39	27,765	0

Table 2: Aggregated Statistics for the Zipf-GOV Benchmark with θ = 0.7 and c=10%

Zipf-GOV c=10%, θ= 0.7	Total # of Bytes	Total Time in ms	Average Recall	Avg Score Error/topKScore	Avg Rank Distance	# Sorted Accesses	# Random Accesses
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 3: Aggregated Statistics for the Zipf-XGOV Benchmark with θ = 0.7 and c=10%

Zipf-XGOV c=10%, θ= 0.7	Total # of Bytes	Total Time in ms	Average Recall	Avg Score Error/topKScore	Avg Rank Distance	# Sorted Accesses	# Random Accesses
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: Aggregated Statistics for the GOV Benchmark with c=10%

GOV c=10%	Total # of Bytes	Total Time in ms	Average Recall	Avg Score Error/topKScore	Avg Rank Distance	# Sorted Accesses	# Random Accesses
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 5: Aggregated Statistics for the XGOV Benchmark with c=10%

XGOV c=10%	Total # of Bytes	Total Time in ms	Average Recall	Avg Score Error/topKScore	Avg Rank Distance	# Sorted Accesses	# Random Accesses
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 6: Aggregated Statistics for the IMDB Benchmark with c=10%

IMDB c=10%	Total # of Bytes	Total Time in ms	Average Recall	Avg Score Error/topKScore	Avg Rank Distance	# Sorted Accesses	# Random Accesses
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

of KLEE for smaller-term queries, (eg 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, 5 and 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 the topKscore. In light of KLEE's strong performance, this is definitely acceptable.

7. Conclusions

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 promotes flexibility. It allows the trading-off of result quality vs performance by: utilizing filters of various sizes at steps 1 and 3, sending various number of

filters in the 1st step, using high, average or low scores of histogram cells for the missing scores in the 1st step, utilizing the cell score upper or lower bounds when determining the interesting columns of CLFM, etc. KLEE even allows the trade-off between bandwidth vs the number of communication phases.

Our comprehensive experiments show that KLEE achieves great performance gains in network bandwidth, query response times, and local peer load, and high quality results. We also developed, implemented, and tested extensions of competing algorithms, which in some cases were found to be better performers than their base algorithm. Again, KLEE was a clear winner. The most appealing performance feature of KLEE is that it introduces the aforementioned great performance benefits consistently, without sensitivity to the key input characteristics (such as score distributions, number of terms/peers, and term correlations).

8. References

[Ag03] S. Agrawal et al.: Automated Ranking of Database Query Results. CIDR 2003

[AKM01] V.N. Anh et al.: Vector-Space Ranking with Effective Early Termination. SIGIR 2001

[Ba05] W.-T. Balke, et al.: Progressive Distributed Top k Retrieval in Peer-to-Peer Networks. ICDE 2005

[BO03] B. Babcock, C. Olston: Distributed Top-K Monitoring. SIGMOD Conference 2003

[Ba03] M. Bawa, et al.: Make it fresh, make it quick: searching a network of personal webservers. WWW 2003

[Bey99] K.S. Beyer, et al.: When Is "Nearest Neighbor" Meaningful? ICDT 1999

[Bl70] B.H. Bloom: Space/Time Trade-offs in Hash Coding with Allowable Errors. Comm. of the ACM, 1970

[BBK01] C. Böhm, S. Berchtold, D.A. Keim: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. 33(3), 2001

[BCG02] N. Bruno, S. Chaudhuri, L. Gravano: Top-k selection queries over relational databases: Mapping strategies and performance evaluation. TODS 27(2), 2002

[BGM02] N. Bruno, L. Gravano, A. Marian: Evaluating Top-k Queries over Web-Accessible Databases. ICDE 2002

[CW04] P. Cao, Z. Wang: Efficient Top-K Query Calculation in Distributed Networks. PODC 2004

[CH02] K.C.-C. Chang, S.-W. Hwang: Minimal probing: supporting expensive predicates for top-k queries. SIGMOD 2002

[CGM04] S. Chaudhuri, L. Gravano, A. Marian: Optimizing Top-K Selection Queries over Multimedia Repositories, TKDE 16(8), 2004.

[Ch04] S. Chaudhuri, et al.: Probabilistic Ranking of Database Query Results. VLDB 2004

[CP02] P. Ciaccia, M. Patella: Searching in metric spaces with user-defined and approximate distances. TODS 2002

[deV02] A.P. deVries, N. Mamoulis, N. Nes, M.L. Kersten: Efficient k-NN Search on Vertically Decomposed Data, SIGMOD 2002.

[Fa99] R. Fagin: Combining Fuzzy Information from Multiple Systems, J. Comput. Syst. Sci. 58(1), 1999

[FLN03] R. Fagin, J. Lotem, M. Naor: Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci. 66(4), 2003.

[Fan98] L. Fan, et al.: A Scalable Wide-Area Web Cache Sharing Protocol, SIGCOMM 1998

[GKB00] U. Güntzer, W. Kießling, W.-T. Balke: Optimizing Multi-Feature Queries for Image Databases. VLDB 2000

[GKB01] U. Güntzer, W. Kießling, W.-T. Balke: Towards Efficient Multi-Feature Queries in Heterogeneous Environments. ITCC 2001.

[HS03] G.R. Hjaltason, H. Samet: Index-driven similarity search in metric spaces. TODS 28(4), 2003.

[Hue05] R. Huebsch, et al.: The Architecture of PIER: an Internet-Scale Query Processor. CIDR 2005

[Kau04] R. Kaushik, et al.: On the Integration of Structure Indexes and Inverted Lists. SIGMOD Conference 2004

[Kou04] N. Koudas, et al.: Approximate NN queries on Streams with Guaranteed Error/performance Bounds. VLDB 2004: 804-815

[LS03] X. Long, T. Suel: Optimized Query Execution in Large Search Engines with Global Page Ordering. VLDB 2003

[MGB04] A. Marian, L. Gravano, N. Bruno: Evaluating Top-k Queries over Web-Accessible Databases. TODS 29(2), 2004

[Na01] A. Natsev, et al: Supporting Incremental Join Queries on Ranked Inputs. VLDB 2001

[NR99] S. Nepal, M. V. Ramakrishna: Query Processing Issues in Image (Multimedia) Databases. ICDE 1999

[PZS96] M. Persin, J. Zobel, R. Sacks-Davis: Filtered Document Retrieval with Frequency-Sorted Indexes, JASIS 47(10), 1996.

[SaLu00] D Salomoni and S. Luitz, "High Performance ThroughputTuning/Measurement" http://www.slac.stanford.edu/grp/scs/net/talk/High_Perf_PPDG_Jul2000.ppt

[So01] A. Soffer, et al: Static Index Pruning for Information Retrieval Systems. SIGIR 2001

[Su03] T. Suel et al.: ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval, WebDB 2003

[TQDFG03] Ajay Tirumala et al.: iPerf: Testing the limits of your network, http://dast.nlanr.net/Projects/Iperf/

[TWS04] M. Theobald, G. Weikum, R. Schenkel: Top-k Query Evaluation with Probabilistic Guarantees. VLDB 2004

[Yu01] C.T. Yu, et al.: Database selection for processing k nearest neighbors queries in distributed environments. JCDL 2001

[YPM03] C.T. Yu, G. Philip, W. Meng: Distributed Top-N Query Processing with Possibly Uncooperative Local Systems. VLDB 2003

[Zi49] G. K. Zipf: Human Behavior and the Principle of Least Effort. Addison-Wesley Press, 1949.