Algebraic Query Optimization for Distributed Top-k Queries

Thomas Neumann, Sebastian Michel
Max-Planck-Institut für Informatik
Saarbrücken, Germany
{neumann, smichel}@mpi-inf.mpg.de

Abstract: Distributed top-k query processing is increasingly becoming an essential functionality in a large number of emerging application classes. This paper addresses the efficient algebraic optimization 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 use a dynamic programming approach to find the optimal execution plan using compact data synopses for selectivity estimation that is the basis for our cost model. The optimized query is executed in a hierarchical way involving a small and fixed number of communication phases. 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.

1 Introduction

Top-k query processing has received great attention for a variety of emerging application classes including multimedia similarity search [GM04, NR99, Fag99, GBK00, BGRS99, NCS$^+$01], preference queries over product catalogs and other Internet sources [CK98, MBG04, BGM02], ranked retrieval of semistructured (XML) documents [GSBS03, TSW05], and data aggregation for "heavy hitters" in network monitoring and other sensor-network applications [CKMS04, CJSS03]. Such queries typically need to aggregate, score, and rank results from different index lists or other forms of data sources. Most of the prior work concentrates on centralized settings where all index lists reside on the same server of server farm. The case where the sources are distributed across a wide area network and the optimization of communication costs is crucial has received only little attention, the most noteworthy exceptions being the TPUT algorithm [CW04], our own prior work on KLEE [MTW05], and the work by [MBG04] and [ZYVG$^+$] both of which build on the family of threshold algorithms (TA) that seems predominant in centralized top-k querying [FLN03, GBK00, NR99].

For the centralized case, the need for choosing good query execution plans from a large space of possible plans has been recognized, and research has started towards a better understanding of algebraic and cost-based optimization of top-k queries. The most prominent, recent example of this line of work is the RankSQL framework [ISA$^+$04, LCIS05], which considers binary trees of outer joins with score aggregation and ranking, so-called "rank joins". The TA family, on the other hand, is a fixed execution strategy with a single m-ary join over precomputed index lists where m is the number of attributes or keywords in the score-aggregation function. For the distributed setting, only fixed algorithms have been proposed without consideration to flexible, data- and/or workload-dependent, construction of different execution plans. For example, TPUT and KLEE, the best known algorithms, first fetch the best k data items from each of the underlying m index lists or data sources, compute the aggregated score s of the rank-k item in this intermediate result, and then issue score-range queries against the different sources to fetch all items with per-attribute scores above s/m. Despite some additional optimizations for better pruning of result candidates, this approach does not explore at all the space of forming different execution trees of joining and aggregating scored items in a flexible manner with attention to networking and local computation costs.

The current paper is the first work that addresses this issue. In contrast to [LCIS05] we consider distributed execution and variable-arity trees. In contrast to [CW04, MTW05] we consider different execution plans, based on estimates of data characteristics, and aim to find the cost-optimal execution plan by algebraic rewriting and a novel way of dynamic-programming-style search for a cost minimum. Compared to the ample work on distributed join queries [MTO99, Kos00], our method is ranking-aware and aims to terminate the query execution as early as possible once the top-k results can be safely identified. More specifically, this paper makes the following research contributions:

- a new model for describing, systematically enumerating, and algebraically rewriting the feasible execution plans in a large space of plans for distributed top-k queries over wide-area networks,
- a new form of dynamic-programming-style optimization for finding the cost-minimal execution plan with cost estimation based on score-distribution statistics,
- a new way of estimating and aggregating statistical score-value distributions, using Poisson mixtures or equi-depth histogram convolutions, to assess the costs of an execution plan, with the salient property that the statistical synopses themselves are very small and thus network-friendly in the optimization phase,
- comprehensive performance studies on large, real-world web data collections, showing that significant performance benefits can be achieved with acceptable overhead of the optimization step itself.

The paper is organized as follows. Section 2 introduces our computational model. Section 3 discusses relevant work on distributed top-k querying and on query optimization in general. Section 4 presents the query optimization techniques together with the actual query processing algorithms. Section 5 presents the cost model and shows how we use distributed statistics to characterize the input data and estimate selectivities. Section 6 presents an experimental evaluation that shows the impact of query optimization on distributed top-k queries. Section 7 concludes the paper and presents ongoing and future work.

2 Computational Model

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). 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, $T = \{t_1, t_2, ..., t_t\}$, 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 $m = t$, 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. However, we will see later that P_{init} can forward the coordination task to another peer that is involved in the query if this is promising to decrease the overall query execution cost. A naive solution would be to have all m cohort peers (i.e. peers that maintain an index list for a specific keyword) 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.

3 Related Work

Top-k query processing has received much attention in a variety of settings such as similarity search on multimedia data [GM04, NR99, Fag99, GBK00, BGRS99, NCS$^+$01], ranked retrieval on text and semi-structured documents in digital libraries and on the Web [AdKM01, LS03, TWS04, KKNR04, BJRS03, PZSD96, YSMQ], network and stream monitoring [BO03, CW04] collaborative recommendation and preference queries on e-commerce product catalogs [YPM03, MBG04, GBK01, CwH02], and ranking of SQL-style query results on structured data sources in general [ACDG03, CDHW, BCG02]. Among the ample work on top-k query processing, the TA family of algorithms for monotonic score aggregation [FLN03, GBK00, NR99] stands out as an extremely efficient and highly versatile method. TA-sorted (aka. NRA) 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 $bestscore(d) < min_k$ can be safely dismissed, where min_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 $bestscore \leq \sum_{i=1...m} high(i)$ 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 min_k value drops below some threshold: $P[worstscore(d) + \sum_i S(i) > min_k] < \varepsilon$, 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.

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 for 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. The recent work by [BNST05] 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. TPUT [CW04] executes TA in three phases: 1) fetch the k best (DocID, Score) entries from each cohort peer and compute the min_k score (i.e., the score of the item currently at rank k) using zero-score values for all missing scores; 2) ask each of the m cohort peers for entries with $Score > min_k/m$, then compute a better min_k value and eliminate candidates whose bestscore is not higher than min_k; 3) fetch the still missing scores for the remaining candidates, asking the cohorts to do random accesses. [YLW$^+$05] presents a modification of TPUT that adapts the min_k/m threshold to the score distributions' peculiarities. KLEE [MTW05] is an approximate version of TPUT that leverages Bloom filter and histograms to prune / filter unpromising documents. [DKR04] considers hierarchical in-network data aggregation where the query hierarchy is given by the network topology. [ZYVG$^+$] presents a threshold algorithm for distributed sensor networks, here, again, the hierarchy is predetermined by the network. [CW04] shortly mentions a hierarchical version of TPUT but does not consider optimized query plans.

Query processing in general is a well studied standard technique. We employ an optimization algorithm that uses a search space exploration technique similar to the Volcano

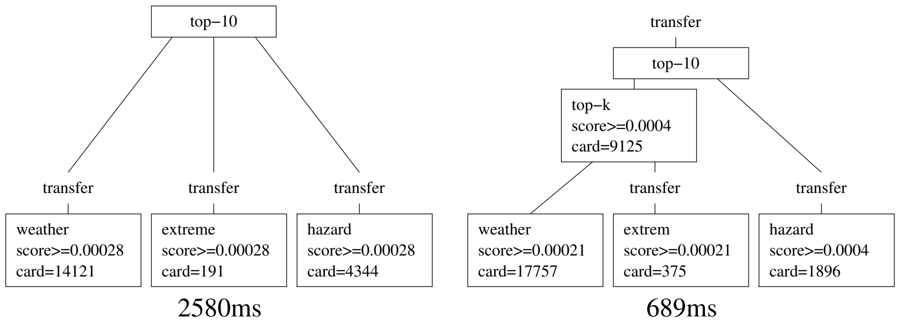


Figure 1: Two equivalent execution plans

approach [GM93]: The search space is explored in a top-down manner by splitting the problem into smaller problems and solving these recursively, memoization prevents duplicate work. The other standard techniques include bottom-up dynamic programming [HFLP89] and transformative optimization [Gra95]. However, only a few paper consider the peculiarities of top-k queries during the optimization phase and treat the actual top-k processing as orthogonal to the main query optimization step. One notable exception is the RankSQL proposal [LCIS05], which integrates the decision about rank computation and usage into the query optimization. It extends the relation algebra towards a rank-aware algebra using an extended search space that includes the available rank aware functions. In contrast to their work, we assume that the basic rank (score) is already known. The problem that arises when dealing with score aggregations is not sufficiently covered in [LCIS05] as the paper is focused on the optimal computation of rank functions. Although they do recognize the problem they only consider scores that follow normal distributions [ISA$^+$04], or they use sampling [LCIS05], which is, from our point of view, prohibitively expensive in a distributed setting.

4 Algorithms

In this paper we use an algorithm that, besides the actual query optimization, essentially consists of the 3-phase algorithm TPUT in an hierarchical environment as shortly mentioned in [CW04]. The query initiator P_{init} retrieves the top-k documents from the involved cohort peers and calculates a top-k estimate by aggregating the scores for the particular documents. min_k denotes the partial score of the document at rank k. In a flat structure, P_{init} would send $\frac{min_k}{m}$ as a threshold to the cohort peers that return all documents above this threshold. In a hierarchical environment, however, the threshold for a particular cohort is based recursively on the threshold of the parent cohort and the number of siblings. For instance, consider the example in Figure 1: In the left query plan, the query initiator sends 0.00028 (i.e. $\frac{min_k}{3}$) to the cohort peers, this refers to an estimated min_k of 0.00084. In the left query plan the P_{init} estimates the same min_k but sends the threshold 0.0004 (i.e. $\frac{min_k}{2}$) to its children. The child at the left receives the threshold 0.0004 and forwards the new threshold $\frac{0.0004}{2}$) to its children. For deeper hierarchies this procedure is applied recursively.

4.1 Overview

We now assume that a query with m terms is started at a peer P_{init}, which has to produce the top-k results, and that the $(docId, score)$ lists are stored at m cohort peers. Note that these peers need not be disjoint, as one peer can handle multiple terms, and P_{init} can also be one of the cohort peers. To model this in the algorithms, we introduce the concept of sites: The basic assumption is that if two index lists are at the same site, they can be accessed simultaneously. Usually a site corresponds to one physical peer. We name the set of all sites relevant for the query S, $|S| \leq m + 1$.

Our algorithms combine the concept of cost based query optimization with the basic execution paradigm of TPUT [CW04]. This means that the execution is done in the following four phases, where the phases 1, 3 and 4 are the same for all algorithms.

1. **Exploration Step**: P_{init} communicates directly with the m cohort peers and retrieves the top-k documents along with statistics and score models that will be used in the selectivity estimation.
2. **Query Optimization**: P_{init} constructs the execution plan according to the selected algorithm.
3. **Candidate Retrieval**: P_{init} sends the execution plan along with the estimated $\frac{min_k}{x}$ thresholds to the involved peers. After having received this information, thus being able to establish connections to children and parent peers the leaf peers start sending $(docId, score)$-pairs to their parents. Subsequently, the former parents, now being treated as children, combine the received information and forward the $(docId, worstscore)$-pairs to their parents, and so on, until the computations has reached the root node. The pruning in the intermediate top-k steps is executed w.r.t. the documents' bestscores. Note that the bestscores are updated during the intermediate merge.
4. **Missing Scores Lookup Phase**: P_{init} constructs the final list of top-k candidates, determines the documents with missing scores, and sends the $docId$s to the corresponding peers. The peers lookup the missing scores and send them to P_{init}, which produces the top-k result.

This phase structure is identical to the TPUT phases (except the optimization phase), and in fact TPUT can be formulated as a special case of this algorithm, as we will see below.

4.2 Algebraic Optimization

Query optimization requires the evaluation of different equivalent execution alternatives. Therefore, the query optimizer has to decide if two alternatives are really equivalent. This is usually done by formulating the query in a formal algebra and using algebraic equivalences. What is different here (in the top-k case) is that these equivalences are somewhat relaxed. The query optimizer also accepts alternatives that are not equivalent in a strict sense, as long as the top-k entries are the same.

This basic premise is shown in Figure 2 and is the idea behind most top-k execution strategies: The problem of finding the top k tuples is reduced to finding all tuples with a sufficient score. Due to the subset relation, more than k tuples can satisfy the condition. This relaxation allows for more efficient evaluation strategies, as reasoning about score is much simpler than reasoning about ranks.

$$\begin{array}{lcl} F_k(S) & = & \text{the first } k \text{ tuples of } S \\ min_k(S) & = & min(\{s.score | s \in F_k(S)\}) \\ \sigma_{\geq t}(S) & = & \{s \in S | s.score \geq t\} \\ \text{top-}k(S) & \subseteq & \sigma_{min_k(S)}(S) \end{array}$$

Figure 2: Rank based vs. score based top-k

$$\begin{array}{lcl} C_k(S_1 \ldots S_n) & = & \cup^{+}_{1 \leq i \leq n} F_k(S_i) \\ min_k(S_1 \ldots S_n) & = & min(C_k(S_1 \ldots S_n) \\ \text{top-}k(S_1 \ldots S_n) & = & \text{top-}k(\cup^{+}_{1 \leq i \leq n} S_i) \\ \text{top-}k(S_1 \ldots S_n) & \subseteq & \cup^{+}_{1 \leq i \leq n} \sigma_{\geq min_k(S_1 \ldots S_n)/n}(S_i) \end{array}$$

Figure 3: Definitions and relation used by TPUT

$$\begin{array}{lcl} \text{top-}k(S_1 \ldots S_n) & \subseteq & \sigma_{\geq min_k(S_1 \ldots S_n)}(\cup^{+}_{1 \leq i \leq n} S_i) \\ \sigma_{\geq t}(S_1 \ldots S_n) & \equiv & \sigma_{\geq t}(\cup^{+}_{1 \leq i \leq l} \sigma_{\geq t/l}(P_i)) \\ \multicolumn{3}{l}{\text{where } P_1 \ldots P_l \text{ is a partitioning of } S_1 \ldots S_n} \end{array}$$

Figure 4: Generalization of TPUT

In practice, the exact (ideal) threshold $min_k(S)$ is not known a-priori and is approximated (conservatively). TPUT does this by sampling the first k entries of each input. The TPUT approach is formalized in Figure 3; we concentrate on aggregation by score summation ($\cup^{+}$), other aggregations schemes can be used analogously. The basic idea is to estimate $min_k(S)$ by fetching some tuples (k from each list) and using the k-th score in the aggregation as a lower bound for $min_k(S)$. This is a safe choice, as at least k tuples above the thresholds have already been seen. Now all tuples in the lists have to have a score of at least $min_k(S)/n$, that can be shown easily.

While this is a nice formalism to get a suitable threshold for aggregation, it offers no optimization opportunities. The threshold propagation is fixed. The TPUT paper [CW04] mentions the possibility of hierarchical TPUT, but offers no formalism. We now provide a formalism and suitable equivalence rules to allow for an algebraic optimization of the TPUT structure. Figure 4 shows the formalism of our approach: First, we do not push the threshold down in a fixed scheme but use a more general transformation as in Figure 2. This is just a minor variation, but essential to give the query optimizer freedom to optimize. More importantly, we present an equivalence that splits a top-k problem into smaller top-k problems that can be solved recursively. This equivalence is used by the query optimizer to find the optimal execution strategy. Note that TPUT itself is a special case of this equivalence, it partitions the input directly into single lists.

4.3 Building Blocks

The previous section presented the algebraic optimization. However, the formulation was very high level. A more detailed look is required to understand the distributed execution, furthermore some logic is required to handle the random lookups in the last phase of the algorithm. We concentrate on the logical operators here, as the corresponding physical operators used during plan generation are mostly obvious. Thus we assume that only one physical operator exists for each logical operator. However, this is not a limitation of the algorithms, they can easily handle alternative physical operators. This could be useful for different evaluation strategies, e.g. for using random lookups in the last phase vs. scanning the lists sequentially.

For each logical operator we specify the optimization rule that is used to create an appropriate plan. As we optimize in a distributed system, the optimizer must be able to ship data to other sites. This is done by the *transfer* operator, the rule checks if the output is at the correct site and ships otherwise:

```
*transfer*($target$,$p$)
    **if** $p$.site=$target$
        **return** $p$
    **return** a plan that ships $p$ to $target$
```

To make the decision about data transfers easier, the algorithms always consider all possible sites in each step. The output of a step is therefore not a single plan, but a set of plans, with one plan for each possible result site. Since it might be cheaper to perform a calculation at one site and ship only the result, the optimization rule *ship* examines all pairs of sites (s_1, s_2) and considers using a *transfer* instead of performing the computation locally:

```
*ship*($plans$)
    **for** $\forall s_1 \in S, s_2 \in S, s_1 \neq s_2$
        $p$=*transfer*($s_2$,$plans[s_1]$)
        **if** $p$.costs<$plans[s_2]$.costs
            $plans[s_1] = p$
    **return** $plans$
```

The top-k operation itself consists of three operations: *base-top-k* scans a list and produces all entries above a min_k threshold (this corresponds to $\sigma_{\geq t}(S)$), *intermediate-top-k* combines intermediate results ($\sigma_{\geq t}(S_1 \ldots S_n)$) and *top-k* produces the final top-k list (the rank based top-k operator). The base rule simply inserts an *base-top-k* operator, it only has to make sure that the score distribution is adjusted according to the min_k:

```
*base-top-k*($min_k$,$t$)
    $p$ = a new plan to scan $t$
    $p.card$ = documents above $min_k$
    $p.scoreDistribution$ = distribution $\geq min_k$
    $plans$ = empty plan set
    $plans$[site of $t$] $= p$
    **return** *ship*($plans$)
```

The intermediate steps combines multiple partial top-k results (either from *base-top-k* or another intermediate step) into a new partial result. The document scores are aggregates, pruned according to the given min_k threshold and the resulting tuples are sorted by descending score. As the intermediate step can combine an arbitrary number of intermediate results the input of this building block is a set of plansets, one planset per input operator. The rule iterates over all sites and combines the plans for each site.

```
*intermediate-top-k*($min_k$,$input$)
    $plans$ = empty plan set
    **for** $\forall s \in S$
        $i = \{p[s] | p \in input\}$
        $h$ = convolution of $\{p.scoreDistribution | p \in i\}$
        $plans[s]$ = a new plan to combine $i$
        $plans[s].card$ = $h$.documents above $mink$
        $plans[s].scoreDistribution$ = part of $h \geq min_k$
    **return** $plans$
```

The final top-k operator takes an intermediate result and constructs the final top-k list by retrieving the missing scores. Note that we only have to consider the unary case, otherwise an intermediate top-k can be used for aggregation.

```
*top-k*($mink$,$plans$)
    $plans$ = empty plan set
    **for** $\forall s \in S$
        $plans[s]$ = a new plan to finalize $plans[s]$
        $plans[s].card$ = $k$
    **return** $plans$
```

4.4 Algorithms

Using these building blocks, the optimization algorithms can be formulated easily. To illustrate the plan construction, we first formulate TPUT for a set of terms T using these constructs:

```
*tput*($T$,$min_k$)
    $b = \{$*base-top-k*$(min_k/|T|, t) | t \in T\}$
    $i =$*intermediate-top-k*$(min_k, b)$
    $p =$*top-k*$(min_k, i)$
    **return** $p[P_{init}]$
```

TPUT reads all lists up to $min_k/|T|$, combines the results and finally calculates the top-k by looking up the missing scores. The last two steps are always executed at P_{init}, therefore the plan for P_{init} is returned.

A simple improvement of the TPUT algorithm is what we call the *simple* optimization algorithm: It behaves like TPUT, but considers performing the aggregation at a different peer than P_{init}. As the final step only produces k tuples, pushing the aggregation down can greatly reduce the costs. However, this decision must be made cost based: Pushing the aggregation down induces additional latency, which can be higher than the gain from the push down. The push-down is done by using the *ship* rule described above.

```
*simple*($T$,$min_k$)
    $b = \{$*base-top-k*$(min_k/|T|, t) | t \in T\}$
    $i =$*intermediate-top-k*$(min_k, b)$
    $p =$*top-k*$(min_k, i)$
    $p$ =*ship*$(p)$
    **return** $p[P_{init}]$
```

A much larger class of execution plans is possible when allowing additional intermediate aggregations. As already mentioned in [CW04], a tree structure can allow for much larger min-k thresholds. However the optimization is also more difficult, especially since an intermediate step can aggregate an arbitrary number of input streams. The optimization is split in two parts: One finds the optimal way to construct intermediate results and the other constructs the optimal top-k after the intermediate results are known.

The intermediate results can be optimized recursively: Given a set of terms, the optimizer recursively solves all partitionings of the terms and combines them to an intermediate result. Dynamic programming (DP) is used to reduce the search space, as term combinations are used multiple times. The DP table maps from $(terms, min_k) \rightarrow (planset)$, i.e., for each term/minimum score combination the optimal plan for each site is stored. Here we use the top-down formulation of dynamic programming (memoization), which naturally follows the recursive optimization scheme:

```
*solveIntermediate*($T$,$min_k$)
    **if** $(T, min_k)$ has already been solved
        **return** the known solution
    **if** $|T| = 1$
        $p$ =*base-top-k*$(min_k, t \in T)$
    **else**
        $p =$ empty plan set
        **for** $\forall P = \{T_i \subset T\}, P$ partitioning of $T$
            $i = \{solveIntermediate(T_i, min_k/|P|)|T_i \in P\}$
            $p'$ =*intermediate-top-k*$(min_k, i)$
            **for** $\forall s \in S$
                **if** $p'[s]$.costs<$p[s]$.costs
                    $p[s] = p'[s]$
        $p$ =*ship*$(p)$
    store $p$ as solution for $(T, mink - k)$
    **return** $p$
```

Note that the algorithm is simplified, see Section 4.5 for performance improvements. The optimal intermediate results can now be used to construct the complete plan quite easily, as the larges intermediate result (all terms) can be used as input for the final top-k operator:

```
*optimal*($T$,$min_k$)
    $i$ =*solveIntermediate*$(min_k, T)$
    $p$ =*top-k*$(min_k, i)$
    $p$ =*ship*$(p)$
    **return** $p[P_{init}]$
```

The *optimal* algorithm considers the whole search space of possible execution trees. In particular, it never produces a worse plan than *tput* or *simple*, as these algorithm produce plans that are inside this search space.

Figure 5 shows an example of an optimized query plan, an output of the above stated optimization technique.

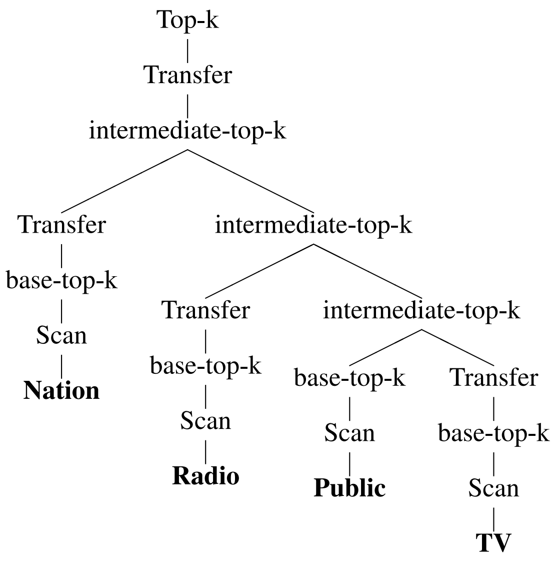


Figure 5: Optimal Execution Plan

4.5 Large Queries

As stated above, the *optimal* algorithm and especially the *solveIntermediate* function considers the whole search space, which, even when memoization is used, means at least $2^{|T|}$ different plans (more because of different min_k values). While this is negligible for common queries (e.g. GOV queries with at most 4 terms), the search space becomes a problem for large queries, e.g. XGOV queries with up to 18 terms.

The search space can be reduced by using cost bound pruning: For a global cost bound the optimizer can use a cheap heuristic (e.g. *simple*) to get a plan quickly, which can be used to prune plans early. During the optimization the optimizer can use already constructed solutions to prune against later ones. In particular, it is possible to find lower cost bounds for a given term set by looking at the original score distributions: If a tuple is already above min_k without convolution it will also be after the convolution. These heuristics allow estimating the costs early, which greatly reduces the search space.

Still the search space grows exponentially, and depending on the implementation and the available hardware the optimal solution will be too expensive to compute for certain queries. Then an idea from the query optimization for joins can be used: Iterative dynamic programming [KS00] performs the full DP computation only for subproblems with a given maximum, chooses one such solution, removes it from the search space and repeats the DP step until the problem is solved. For the top-k processing this means that optimal intermediate results for up to a given size are computed, then one solution is chosen (we used the cheapest one) and the terms occurring in this solution are considered solved and replaced by a new pseudo-term. This is repeated until the problem is solved. While this does not guarantee the optimal solution, it results in quite good plans in practice (and of course the optimal solution is still possible).

5 Distributed Statistics and Cost Model

Using query optimization techniques for distributed top-k retrieval assumes that some execution alternatives are better than others. This can be formalized using a cost model, which describes how many resources are used by each alternative. This requires accurate cardinality estimations to find the optimal execution plan. While this is always the case for query optimization in general, the cardinalities during top-k processing depend on the scores of the computed data. Therefore standard cardinality estimators are not enough, the query optimizer needs to estimate score distributions. Prior work on this issue either used fairly crude models like assuming Normal distributions [ISA$^+$04], which is not a good fit for real-data scores, or required extensive computations like sampling of histogram maintenance [TWS04] that may incur high costs in a distributed setting with a high-latency network.

When optimizing distributed queries, there exist mainly two interesting optimization goals: Either minimize the query response time or minimize the total resource consumption, especially the network bandwidth. Minimizing the query response time is obviously relevant, as someone (either a user or a program) is waiting for the answer. Minimizing the network bandwidth is also interesting, as it allows for more parallel queries. The algorithms presented in the next section can handle both goals and we show experimental results for both in Section 6, but we concentrate on minimizing response time.

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.

In this work we consider Poisson Mixture Models and Equi-Depth Histograms to model score distributions for single-term index lists. Score distributions of multi-term index lists (i.e. combined single-term index lists using score aggregations) are computed as convolutions [All90].

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

$$\pi(k, \theta) = \frac{e^{-\theta} * \theta^k}{k!}$$

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

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 [CG]. In our work we use Two Poisson Mixes to describe the score distribution for each index list in an ultra compact way as each Two Poisson Model requires only 3 floating point numbers so that the additional network resource consumption is negligible. The *Two Poisson Model* is a simple example of a Poisson mixture:

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

[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.

However, in general, Poisson Mixture Models suffer from the inability to capture extreme variations in the score distributions. Histograms are a better tunable approach to represent score distributions but the convolution of equi-depth histograms is computationally expensive and they cause higher network traffic than Two Poisson Mixes, albeit providing a more accurate estimation so that we are able to trade off accuracy vs efficiency.

For Poisson Mixes, and Histograms the quality of the convolution w.r.t. to the score distribution of the true data depends to a large extent on the cardinality of the intersection of the involved data. Without detailed knowledge of the cardinality of the resulting data, the score distribution model will be way off, thus the query optimizer cannot work properly. For instance, recall that the convolution of two Poisson distributions is a Poisson distribution with the mean being the sum of the two means of the original distributions. In an extreme case the underlying data of the two distributions is nearly disjoint, thus the mean of the convolution is not at all equal to the sum of means. As adequate score distribution models are an essential part of an effective query optimizer, we have focused on integrating information about the input data's mutual correlation into the convolution of the data's score distribution. In particular, we are interested in the cardinality of the intersection (union) of two data sets (i.e. index lists). Estimating overlap of sets has been receiving increasing attention for modern emerging applications, such as data streams, internet content delivery, etc. In prior work [MBTW06] we have conducted a comprehensive evaluation of Bloom Filters [Blo70], Hash Sketches [FM85], and min-wise independent permutations (MIPs) [BCFM00] that show that MIPs are best suitable for our purpose.

Now the query optimizer uses these statistics to estimate the number of tuples above a certain threshold. When only considering base relations (i.e. unaggregated data), this estimation can be done directly by using the available score distribution. The only point to keep in mind is that the threshold affects the score distribution, as all tuples below the threshold are missing afterwards. But this can be implemented easily. More interesting is the score distribution of aggregated data: Assuming independence, the score distribution of the sum of two scores can be estimated by a simple convolution of the score distributions of the summands: $P_{S_1+S_2}(x) = \Sigma_{i+j=x} P_{S_1}(i) * P_{S_2}(j)$

However this estimation is only correct if the scores are indeed summed up. Here, this means that the estimation assumes that every tuple in S_1 finds a matching tuple in S_2. If this not the case (e.g. if S_1 and S_2 are disjoint), the score is overestimated. We takes this effect into account by using the intersect and union estimations provided by MIPs: Only the tuples in the intersection are convoluted, for the others the score distribution is unchanged. This lead us to the following estimation:

$$\begin{aligned} P_{S_1+S_2}(x) &= \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}[\Sigma_{i+j=x} P_{S_1}(i) * P_{S_2}(j)] + \\ & (1 - \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}) \frac{|S_1| P_{S_1}(x) + |S_2| P_{S_2}(x)}{|S_1| + |S_2|}. \end{aligned}$$

In our experiments this estimation predicts the actual score distribution very well. Taking the size of the intersection into account is important, as it encourages intersecting relatively disjoint sets early, which can greatly reduce the output cardinality.

6 Experimentation

6.1 Setup and Test Data

Our implementation of the testbed and the related algorithms was written in C++. 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. Two real-world data collections were used in our experiments: GOV and IMDB. 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). We used 50 queries from the Web Track's distillation task where each query consists 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/$\sim$wn/). The expansion resulted in queries with, on average, twice as many terms, with the longest query containing 18 terms. For instance, the GOV query *juvenile delinquency* has been expanded into *juvenile delinquency youth minor crime law jurisdiction offense prevention*. 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. The IMDB queries contain text and structured attributes.

6.2 Competitors

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, so we report only the results for the original TPUT version.
Simple: This algorithm is essentially the same as TPUT but allows a moving coordinator.
Optimal: As one of our key contributions is to show the suitability and significant benefits of optimizing hierarchical top-k algorithms, we implemented a hierarchical version of TPUT. This algorithm essentially consists of multiple TPUT instances organized in a tree like structure.

6.3 Metrics

Network Bandwidth Consumption: This represents the total number of bytes transferred during query execution. It includes the bytes for the actual data items and the size of the required statistics as described in Section 5.

Query Response Time: This represents the elapsed, ”wall-clock”time for running the benchmarks. It includes network time, local i/o time, and time for query optimization and local processing. Our optimization is primarily focused on minimizing the query response time but as experiments indicate the overall network traffic will decrease too, since the network traffic is an essential part of the overall response time.

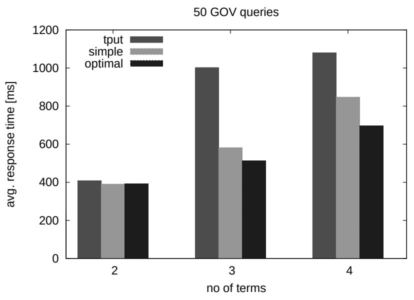
6.4 Experiments

We report on experiments performed for each of the benchmarks, GOV, XGOV, IMDB. In all experiments queries are for the top-10 results. 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 the execution is running in parallel, each operator in the execution plan has to wait for the slowest of its inputs.

Note that the execution time and network consumption includes the transfer of the required statistical data. While the TPUT algorithm does not require statistics, the other algorithms ship the histograms (or poisson mixes) and MIPs during Phase 1. This overhead is included in the results.

6.5 Performance Results

Figure 6(left) shows the average query response time for the GOV benchmark where queries with the same number of query terms were grouped together. For the two-term queries there is almost nothing to gain as there are not many alternatives for the execution plan. However, for the three- and four-term queries we clearly outperform TPUT. Figure 6(right) shows the total network traffic for the GOV benchmark. We clearly beat TPUT for all numbers of query terms; TPUT causes two times more network traffic than the optimized query execution, and even the simple approach shows a pretty strong improvement. Recall at this point that we optimize the query response time and not the network resource consumption, thus we can observe a bigger gain in query response time than in overall network traffic, as can be seen in the Figures.



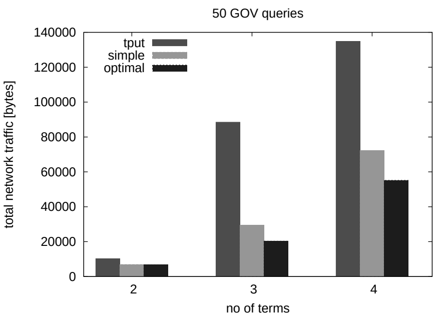
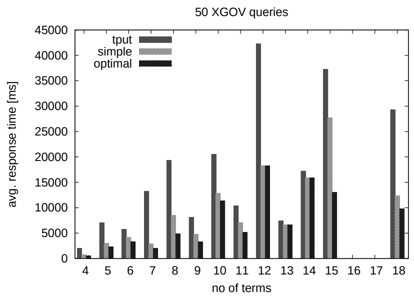


Figure 6: Query Response Times (left) and Total Network Traffic (right) for the GOV Benchmark

Figure 7(left) shows the average response time for the XGOV benchmark. We clearly outperform TPUT for all number of query terms. Note that the XGOV benchmark does not contain queries with 16 or 17 terms. Figure 7(right) shows the total network traffic for the XGOV benchmark. Although we optimize the queries with respect to the overall response time the total network traffic is also smaller.



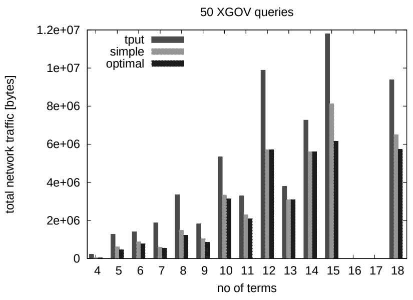
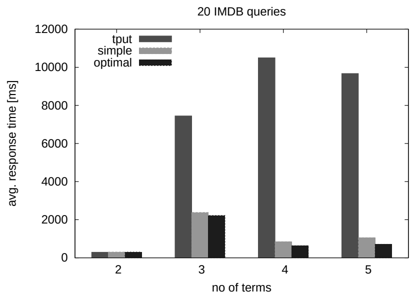


Figure 7: Query Response Times (left) and Total Network Traffic (right) for the XGOV Benchmark

Figure 8(left) shows the query response time for the IMDB benchmark. Similar to the GOV benchmark, all algorithms show nearly the same performance for the two keyword queries. However, for queries with three or more keywords the optimization has remarkable performance gains. For one particular query we gain a factor of around 32 for the optimal plan. The median factor of the performance-improvement is around 5. We observe a similar behavior for the overall network traffic as shown in Figure 8(right).



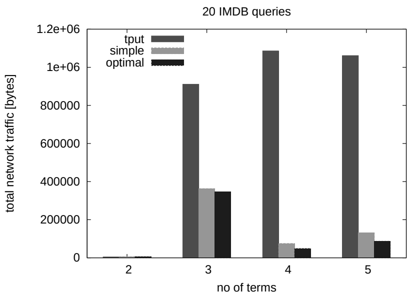


Figure 8: Query Response Time (left) and Total Network Traffic (right) for the IMDB Benchmark

To compare the usefulness of the Histogram based query optimization with the Two Poisson Model based optimization, Figure 9 show the relative performance gains in query response time of the optimized query plans for these two score distribution models for the XGOV benchmark. Obviously, as expected, the histogram based optimization shows a better performance than the Poisson based approach. As discussed above, Histograms offer a more accurate way to represent an index list's score distribution, and although Poisson Models are more compact, the overhead for the histograms is nearly negligible too. As mentioned in Section 5, (Two) Poisson Models cannot capture extreme variations in the score distribution that were present in our data. In particular, the IMDB index lists have an extremely skewed score distribution with ties. The scores from the GOV collection were based on $tf * idf$ that are skewed too. A more smoothed scoring function like BM25 [RW94] would have been better suitable for the Two Poisson Model. However, Poisson Mixes give reasonably good approximations and are interesting as they have extremely small overhead.

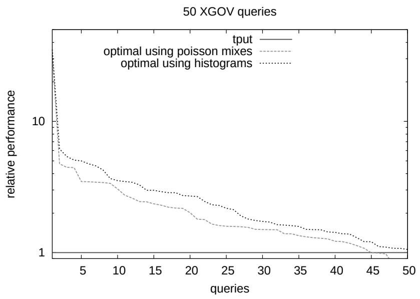


Figure 9: Comparison of Histogram and Two Poisson Model based Optimization for the XGOV Benchmark.

7 Conclusion

This paper has addressed efficient algebraic optimization of top-k queries in a distributed environment. We have shown how to deal with score-threshold based operators during query optimization. The proposed algorithm uses compact data synopsis, shipped at query execution time to find the optimal query execution plan for a particular information need. The experimental evaluation shows that the additional resource consumption caused by the query optimization is nearly negligible and that the optimized queries are superior to the standard top-k queries both in terms of network resource consumption and query response time. Future work includes the integration of other distributed top-k algorithms than [CW04] and multi-query optimization across different distributed top-k queries.

References

[ACDG03] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das und Aristides Gionis. Automated Ranking of Database Query Results. In *CIDR*, 2003.

[AdKM01] Vo Ngoc Anh, Owen de Kretser und Alistair Moffat. Vector-Space Ranking with Effective Early Termination. In *SIGIR*, 2001.

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

[BCFM00] Andrei Z. Broder, Moses Charikar, Alan M. Frieze und Michael Mitzenmacher. Min-Wise Independent Permutations. *Journal of Computer and System Sciences*, 60(3), 2000.

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

[BGM02] Nicolas Bruno, Luis Gravano und Amélie Marian. Evaluating Top-k Queries over Web-Accessible Databases. In *ICDE*, 2002.

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan und Uri Shaft. When Is "Nearest Neighbor" Meaningful? *Lecture Notes in Computer Science*, 1999.

[BJRS03] M. Bawa, R. Jr, S. Rajagopalan und E. Shekita. Make it Fresh, Make it Quick – Searching a Network of Personal Webservers, 2003.

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

[BNST05] Wolf-Tilo Balke, Wolfgang Nejdl, Wolf Siberski und Uwe Thaden. Progressive Distributed Top k Retrieval in Peer-to-Peer Networks. In *ICDE*, 2005.

[BO03] Brian Babcock und Chris Olston. Distributed Top-K Monitoring. In *SIGMOD*, 2003.

[CDHW] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis und Gerhard Weikum. Probabilistic Ranking of Database Query Results. In *VLDB 2004*.

[CG] K. Church und W. Gale. Poisson mixtures. In *Natural Language Engineering, 1(2), 1995.*

[Cha02] Soumen Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann, San Francisco, 2002.

[CJSS03] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck und Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.

[CK98] Michael J. Carey und Donald Kossmann. Reducing the Braking Distance of an SQL Query Engine. In *VLDB*, 1998.

[CKMS04] Graham Cormode, Flip Korn, S. Muthukrishnan und Divesh Srivastava. Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-Dimensional Data. In *SIGMOD*, 2004.

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

[CW04] Pei Cao und Zhe Wang. Efficient top-K query calculation in distributed networks. In *PODC '04*, 2004.

[CwH02] Kevin Chen-Chuan Chang und Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, 2002.

[DKR04] Antonios Deligiannakis, Yannis Kotidis und Nick Roussopoulos. Hierarchical In-Network Data Aggregation with Quality Guarantees. In *EDBT*, 2004.

[Fag99] Ronald Fagin. Combining Fuzzy Information from Multiple Systems. *J. Comput. Syst. Sci.*, 58(1), 1999.

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

[FM85] Philippe Flajolet und G. Nigel Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2), 1985.

[GBK00] Ulrich Guntzer, Wolf-Tilo Balke und Werner Kiesling. Optimizing Multi-Feature Queries for Image Databases. In *VLDB Journal*, 2000.

[GBK01] Ulrich Güntzer, Wolf-Tilo Balke und Werner Kießling. Towards Efficient Multi-Feature Queries in Heterogeneous Environments. In *ITCC*, 2001.

[GM93] Goetz Graefe und William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, 1993.

[GM04] Luis Gravano und Amelie Marian. Optimizing Top-k Selection Queries over Multimedia Repositories. In *TKDE*, 2004.

[Gra95] Goetz Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3), 1995.

[GSBS03] L. Guo, F. Shao, C. Botev und J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

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

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman und Hamid Pirahesh. Extensible Query Processing in Starburst. In *SIGMOD*, 1989.

[ISA$^+$04] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter und Ahmed K. Elmagarmid. Rank-aware Query Optimization. In *SIGMOD*, 2004.

[KKNR04] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton und Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD*, 2004.

[Kos00] Donald Kossmann. The State of the art in distributed query processing. *ACM Comput. Surv.*, 32(4), 2000.

[KS00] Donald Kossmann und Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. In *TODS*, 25(1), 2000.

[LCIS05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas und Sumin Song. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In *SIGMOD*, Seiten 131–142, 2005.

[LS03] Xiaohui Long und Torsten Suel. Optimized Query Execution in Large Search Engines with Global Page Ordering. In *VLDB*, 2003.

[MBG04] Amélie Marian, Nicolas Bruno und Luis Gravano. Evaluating top-*k* queries over web-accessible databases. In *TODS*, 29(2), 2004.

[MBTW06] Sebastian Michel, Matthias Bender, Peter Triantafillou und Gerhard Weikum. IQN Routing: Integrating Quality and Novelty for Web Search. In *EDBT*, 2006.

[MTO99] Patrick Valduriez: M. Tamer Ozsu. *Principles of Distributed Database Systems*. Prentice-Hall, 1999.

[MTW05] Sebastian Michel, Peter Triantafillou und Gerhard Weikum. KLEE: A Framework for Distributed Top-k Query Algorithms. In *VLDB*, 2005.

[NCS$^+$01] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li und Jeffrey Scott Vitter. Supporting Incremental Join Queries on Ranked Inputs. In *The VLDB Journal*, 2001.

[NR99] Surya Nepal und M. V. Ramakrishna. Query Processing Issues in Image (Multimedia) Databases. In *ICDE*, 1999.

[PZSD96] Michael Persin, Justin Zobel und Ron Sacks-Davis. Filtered Document Retrieval with Frequency-Sorted Indexes. *Journal of the American Society of Information Science*, 47(10), 1996.

[RW94] Stephen E. Robertson und Steve Walker. Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval. In *SIGIR*, 1994.

[SL00] D. Salomoni und S. Luitz. High Performance Throughput Tuning/Measurement. http://www.slac.stanford.edu/grp/scs/net/talk/High_Perf_PPDG_Jul2000.ppt. 2000.

[SMwW$^+$03] Torsten Suel, Chandan Mathur, Jo wen Wu, Jiangong Zhang, Alex Delis, Mehdi Kharrazi, Xiaohui Long und Kulesh Shanmugasundaram. ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval. In *WebDB*, 2003.

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

[TSW05] Martin Theobald, Ralf Schenkel und Gerhard Weikum. An Efficient and Versatile Query Engine for TopX Search. In *VLDB*, 2005.

[TWS04] Martin Theobald, Gerhard Weikum und Ralf Schenkel. Top-k Query Evaluation with Probabilistic Guarantees. In *VLDB*, 2004.

[YLW$^+$05] Hailing Yu, Hua-Gang Li, Ping Wu, Divyakant Agrawal und Amr El Abbadi. Efficient Processing of Distributed Top- Queries. In *DEXA*, 2005.

[YPM03] Clement T. Yu, George Philip und Weiyi Meng. Distributed Top-N Query Processing with Possibly Uncooperative Local Systems. In *VLDB*, 2003.

[YSMQ] Clement Yu, Prasoon Sharma, Weiyi Meng und Yan Qin. Database selection for processing k nearest neighbors queries in distributed environments. In *JCDL '01*.

[ZYVG$^+$] D. Zeinalipour-Yazti, Z. Vagena, D. Gunopulos, V. Kalogeraki, V. Tsotras, M. Vlachos, N. Koudas und D. Srivastava. The threshold join algorithm for top-k queries in distributed sensor networks. In *DMSN '05*.