Distrib Parallel Databases (2009) 26: 3–27
DOI 10.1007/s10619-009-7041-z

Distributed top-*k* aggregation queries at large

Thomas Neumann · Matthias Bender · Sebastian Michel · Ralf Schenkel · Peter Triantafillou · Gerhard Weikum

Published online: 18 June 2009

Abstract Top-*k* query processing is a fundamental building block for efficient ranking in a large number of applications. Efficiency is a central issue, especially for distributed settings, when the data is spread across different nodes in a network. This paper introduces novel optimization methods for top-*k* aggregation queries in such distributed environments. The optimizations can be applied to all algorithms that fall into the frameworks of the prior TPUT and KLEE methods. The optimizations address three degrees of freedom: 1) hierarchically grouping input lists into top-*k* operator trees and optimizing the tree structure, 2) computing data-adaptive scan depths for different input sources, and 3) data-adaptive sampling of a small subset of input sources in scenarios with hundreds or thousands of query-relevant network nodes. All optimizations are based on a statistical cost model that utilizes local synopses, e.g., in the form of histograms, efficiently computed convolutions, and estimators based on

T. Neumann (✉) · M. Bender · G. Weikum
Max-Planck-Institut für Informatik, Saarbrücken, Germany
e-mail: neumann@mpi-inf.mpg.de

M. Bender
e-mail: mbender@mpi-inf.mpg.de

G. Weikum
e-mail: weikum@mpi-inf.mpg.de

S. Michel
École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
e-mail: sebastian.michel@epfl.ch

R. Schenkel
Saarland University and Max-Planck-Institut für Informatik, Saarbrücken, Germany
e-mail: schenkel@mpi-inf.mpg.de

P. Triantafillou
University of Patras, Patras, Greece
e-mail: peter@ceid.upatras.gr

order statistics. The paper presents comprehensive experiments, with three different real-life datasets and using the ns-2 network simulator for a packet-level simulation of a large Internet-style network.

Keywords Top-*k* · Distributed queries · Query optimization · Cost models

1 Introduction

1.1 Motivation and problem statement

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

These issues have been intensively researched in recent years (e.g., [7, 10, 15, 20, 21, 28, 35, 40]), and are now fairly well understood for a centralized setting with all data residing on the same server. The current state-of-the-art algorithms for distributed top-*k* querying [4, 9, 33, 44] address the peculiarities of a distributed setting (in particular communication cost), but fall short of being a perfect solution for really large-scale distributed settings (e.g., highly decentralized and dynamic peer-to-peer systems), where even other performance issues become critical and require different compromises. This paper develops novel techniques to address the peculiarities of such large-scale systems and shows their practical viability.

Conceptually, the data we consider resides in a (virtual) table that is horizontally partitioned across many nodes in a wide-area network; partitionings are typically along the lines of value ranges, creation dates, or creators. The queries that we want to evaluate on the (virtual) union of all partitions compute the top-*k* globally most frequent, least frequent, or highest scoring items across the entire network. Further, we assume a monotonic aggregation function, such as most of the popular aggregation functions (maximum, minimum, (weighted) summation).

This framework has important real-world applications:

- Network monitoring over distributed logs [19]. Items are IP addresses, URLs, or file names in P2P file sharing, and queries could aggregate occurrence frequencies or transferred bytes.
- Sensor networks with sensors that have local storage and are periodically polled [31]. Possible items are chemicals that contribute to water or air pollution, and the values represent actual measurements of their concentration. Typical aggregations are based on specific time periods (e.g., morning hour vs. evening hour).

– Mining of social communities and their behavior [18]. Typical items are specific user groups. Interesting aggregations consider frequencies of postings to different blogs, or "social tags" and ratings assigned to user-created content, or statistical information from query logs and click streams.

1.2 Computational model, assumptions

Following [9, 33, 44], we consider a distributed system with m peers P_j, $j = 1, \ldots, m$. It is assumed that every node can communicate with every other node – possibly with different network costs, but without any limitation of functionality. This can, if necessary, be assured by means of "proxy" nodes. Each peer P_j owns a fragment of an abstract relation, containing items I and their corresponding (local) values $v_j(I)$. Such pairs are accessible at each peer P_j in sorted order by descending value, i.e., in a (physically or virtually) sorted list L_j. These lists can be implemented by materializing local index lists, but other ways are conceivable, too. Notice that an item can, and usually does, appear in the lists of more than one peer. Often, some popular items (e.g., URLs or IP addresses in a network traffic log) appear in the lists of nearly all peers.

A query $q(k)$, initiated at a peer P_{init}, aims at finding the k items with highest aggregated values $V(I) = Aggr_{P_j} v_j(I)$ over all peers P_j. For the sake of concreteness, we will use summation for value aggregation throughout the paper, but weighted sums and other monotonic functions are supported analogously. Scanning the local list L_j allows each peer P_j to retrieve and ship a certain number of its locally highest-value items. The receiving peer (e.g., P_{init}) can then employ a threshold algorithm [20, 21, 35] for value aggregation and determining whether previously unseen result candidates potentially qualify for the final top-k result, or if deeper scans or further probings of unknown values are needed to safely eliminate result candidates.

All algorithms in this paper proceed in rounds [9, 33, 44]: in each round, requests are sent to certain network nodes to either scan local lists to a certain depth or to probe for an item's local value. The requestor subsequently collects and aggregates the results and updates its bookkeeping about top-k candidates. The most important resource to optimize is communication bandwidth, or equivalently, the number of item-value entries that are shipped over the network. In addition, but as secondary criteria, we also observe message latencies and processing loads incurred at the nodes.

This work sets aside node failures during query execution. In case of temporary node failures or nodes leaving the system, we can adopt the method of [1], which proposes to send partial results directly to the query initiator, or we can apply a reorganization step for the affected portion of the query execution plan.

1.3 Contribution and outline of the paper

A standard way of performing distributed top-k aggregation queries is illustrated in Fig. 1 (a). The figure shows four input lists on four different peers and the message flow to a fifth peer (p_0) that has posed a top-k query. The four lists have different sizes, and we assume that the query processing uses a uniform value threshold of 0.3 for its scan depth. We will later contrast this execution plan with better ones based

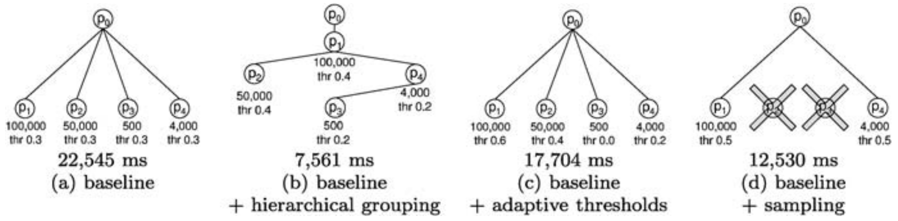

Fig. 1 Execution plans illustrating the optimization techniques

on our methods. Figure 1 also shows response times measured in our testbed, as anecdotic evidence of our performance gains.

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

– The flexible formation of hierarchical groups of node subsets that are considered together. This divide-and-conquer paradigm (cf. Fig. 1 (b)) avoids overly broad top-*k* aggregation queries that involve too many nodes at the same time and could lead to (incoming) bandwidth bottlenecks at the root of the aggregation. On the other hand, it introduces the combinatorial problem of choosing appropriate groups and forming a tree of cascaded top-*k* operators (possibly with different *k* at different stages). We provide exact methods and heuristic approximations for solving this optimization.
– While previous methods have usually propagated uniform scan depth thresholds to other peers, we propose an adaptive method for choosing different scan depth thresholds at different nodes, driven by the statistical information about the value distributions in the local lists (cf. Fig. 1 (c)).

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

– Choosing a sufficiently small subset of nodes as samples, based on a statistical error estimation (cf. Fig. 1 (d)). The sample contains nodes that are most likely to contribute the highest values to the top-*k* aggregation. Depending on the estimated error, the sample can optionally be increased in an additional round, or a small number of top-*k* candidate items may be probed at all network nodes.

All three techniques are based on a statistical cost predictor, which is also a contribution of this paper:

– Estimating the costs of the considered groupings, scan depths, or samples, based on concisely approximating local value distributions by histograms, computing convolutions to combine multiple histograms, and using results from order statistics to estimate rank-*k* values.

The paper presents an extensive evaluation, based on three different real-life datasets and realistic workloads, to demonstrate the scalability of our approaches and their superior performance compared to prior work. The underlying network is simulated by the ns-2 network simulator, a highly detailed and validated model for Internet traffic, widely used in the networking community.

The paper is based on but significantly extends our earlier work in [37]. The main novel contributions of this paper include (1) the sampling technique, (2) proofs of the theorems, (3) some discussion of the effects of network dynamics, and (4) experiments with a third data set.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 4 introduces our cost prediction model. The hierarchical grouping technique and its optimization is presented in Sect. 5. Section 6 presents the technique for adaptive scan depths, before Sect. 7 discusses our sampling methods. Section 9 presents a comprehensive experimental evaluation of our techniques. We briefly study the effects of network dynamics in Sect. 8.

2 Related work

Top-k query processing has received much attention in a variety of settings such as similarity search on multimedia data [12, 20, 34, 35, 41], ranked retrieval on text and semi-structured documents in digital libraries and on the Web [2, 26, 29, 30, 40], network and stream monitoring [3, 9, 16, 27], collaborative recommendation and preference queries on e-commerce product catalogs [10, 22, 32], and ranking of SQL-style query results on structured data sources in general [7, 11, 23, 39, 43]. Within this rich body of work, the *TA* (threshold algorithm) family for monotonic score aggregation [20, 21, 35] has proven to be an extremely efficient and highly versatile method. It comes in variants with sequential scans of index lists only (NRA), or with a flexible combination of sorted and random accesses (CA). Another interesting approach in this context is the RankSQL work [28] that incorporates query optimization techniques into top-k processing in relational database systems. RankSQL includes a technique for estimating the score of the kth result of a top-k query by sampling a small subset of the items; we propose a technique in Sect. 7 that estimates this score from statistics on the score distribution in the lists only, as RankSQL-style sampling would require contacting each peer in advance. Techniques to estimate the number of items read from different lists using statistical summaries, similar to those introduced in Sect. 4.1, have also been used in [23, 38] in the context of optimizers for ranking queries.

The first distributed TA-style algorithm for top-k queries over Internet data sources has been proposed by Bruno et al. [8, 32]. This hybrid algorithm allows both sorted and random access to input lists but tries to avoid random accesses depending on the access costs and limitations of the data sources. Scheduling strategies for random accesses to expensive data sources were also addressed in [10] for a setting with centralized sorted accesses. Zhang and Suel [46] consider distributed variants of TA with sorted accesses only, continuously sending parts of the lists between the network nodes until the top-k answers have been found, where the number of communication steps is limited only by the size of the shortest list. This approach emphasizes the goal of minimizing the total work for scanning input lists on the underlying nodes, but it disregards communication costs as the decisive cost factor.

In contrast, state-of-the-art algorithms for distributed top-k aggregation use a fixed number of communication rounds to bound latency and aim to minimize the total network bandwidth consumption. The first algorithm in this family is *TPUT* (Three-Phase Uniform Threshold) [9], where a query coordinator, typically the query initiator, executes a three-phase distributed threshold algorithm by retrieving some data items from all nodes, computing a threshold ($min\text{-}k/m$) from the observed data, and then using a range query for all data items above the threshold to get all results candidates. See Sect. 3 for a more detailed discussion. *TPAT* [44] is a modification of TPUT where the $min\text{-}k/m$ threshold is adapted to the specifics of the value distributions; however, the authors state that their solution may incur very high computational cost.

KLEE [33] is a framework for distributed top-k processing that utilizes a combination of histograms and Bloom filters to reduce the communication costs of TPUT-style algorithms. When a node is requested to return its locally best items, a KLEE node piggybacks a histogram of the local value distribution and also Bloom filters as compact synopses of the items for each of the top-c histogram cells or groups of consecutive cells (where c is a tunable parameter). The receiver of these synopses, usually the query originator, can combine the Bloom filters from different network nodes for an approximate aggregation of values. The additional information obtained from the per-cell synopses often allows the query processor to derive a higher $min\text{-}k$ threshold than TPUT would have; the subsequent round(s) of retrieving all list entries with value above $min\text{-}k/m$ is more restrictive and can save communication as well as processing costs.

[1] introduces an elegant top-k algorithm for unstructured peer-to-peer systems with epidemic messaging (aka. flooding). In contrast to our computational model, it is assumed that each final recipient of the query message executes the full query – so there is no aggregation on the return path.

Specific network topologies are considered in [4, 45], leading to optimizations for hypercube or tree topologies. Unlike these approaches, TPUT and KLEE have been designed for general networks without any assumptions on network topology. The optimization techniques introduced in this paper are applicable to any algorithm of the TPUT or KLEE families.

3 Query processing framework

Before discussing our optimization techniques, we briefly describe the basic query processing primitives we rely on. We first discuss the assumed setting and the original TPUT algorithm [9], and then expand this to KLEE and further primitives required for our algorithm.

As scenario we assume where that the data distributed over multiple nodes in an Internet-style network with full connectivity. A user now what to execute an aggregation query originating in a certain node, i.e., the query result must be delivered to the querying node. As a first step, the querying node must decide which nodes are relevant for the current query. For some kinds of queries (e.g., network monitoring) this can be all nodes, but in general only some of the nodes (perhaps 10–100) will qualify for a given query. Note that determining the set of relevant nodes is a non-trivial problem in itself, but is beyond the scope of this work. See for example [5] for a discussion. In the following we only consider the relevant nodes. After this selection step, the query originator executed the top-k aggregation query on all relevant nodes.

Algorithmic details vary, but the basic approach in distributed top-k processing is usually similar to the TPUT algorithm, which operators in three phases:

- Phase 1: Retrieve the top-k list entries from each of m network nodes and compute the k-th largest aggregated value of these items (*min*-k), assuming a value of 0 for all unknown values (i.e., values in lists where the item has not yet been seen).
- Phase 2: Revisit all m nodes and ask for all list items with $value \geq min\text{-}k/m$, then recompute *min*-$k$, and eliminate candidates which cannot qualify anymore for the global top-k items.
- Phase 3: Retrieve all missing values for the remaining top-k candidate items by random accesses to the input lists where the items have not yet been seen.

The TPUT algorithm is exact, i.e., it always computes the correct top-k result: After the first phase it has seen at least k distinct items, thus *min*-k is the lower bound for the score of the k-th item in the final result. In the second phase it retrieves all items with a value $\geq \frac{min\text{-}k}{m}$, which means that an item not seen in this phase must have all local scores $< \frac{min\text{-}k}{m}$, and consequently their total score must be *min*-k (assuming summation). Therefore the algorithm has seen all possible top-k results in the first two phases, phase three is just needed to retrieve missing partial scores and to finalize the result order.

The KLEE algorithm [33] is structurally similar to TPUT. It uses probabilistic pruning, and passes additional information (histograms and bloom filters) during query processing to eliminate data items, but the basic data flow is similar. KLEE is not exact, though, and it does no perform score lookups in the last phase to construct the final sort order but relies on the information already available.

Our optimization techniques adapt these algorithms in different ways to improve query processing times (see the following sections). We do no change the underlying principles, though. For example we modify the TPUT thresholds to improve query performance, but we preserve the exactness of TPUT. Similarly, we adapt the data flow to a hierarchical strategy to improve query performance, but the underlying item processing uses the same upper-bound/lower-bound pruning as the original TPUT algorithm.

Note that in the following sections we make some assumptions about score distributions within peers, availability of (small) histograms for these scores etc. These assumptions only affect the query processing times, not the correctness of the algorithm. If we do not have any information the algorithm implicitly falls back to uniform thresholds, i.e., the original TPUT strategy. Similar for assumptions about statistical independence etc. While these assumptions do not perfectly hold in practice due to correlations and data skew, they allow us to make optimization decisions that improve query processing without changing the result. (This is not strictly true for the approximate algorithms, as here a misestimation can have more severe consequences. We evaluate the result quality in Sect. 9).

4 Cost prediction

All cost-based optimizations rely on cost predictions to decide which execution strategy is preferable. We now discuss two prediction primitives required for our top-k optimization. First, we must predict the number of items with a score above a threshold, as this affects the behaviour of the top-k algorithms. Second, we must predict the cost of the resulting network transfers, as they affect the observable runtime. We will discuss more advanced estimation methods in Sect. 7.

4.1 Estimating the number of transfered items

An important building block for our algorithms is the estimation of the number of items transfered in the second phase of the distributed execution. For the original TPUT and KLEE algorithms, this corresponds to estimating the number of items at each peer with a score at least $\mathit{min}\text{-}k/m$. In our advanced algorithms, we additionally need to estimate the number of items whose aggregated score from a subset of all lists is above a threshold.

To perform these estimations, each peer computes summaries of the lists that it stores and distributes them to all other peers. This distribution can be done at a very small overhead in the first phase of the algorithms, when each peer needs to be contacted anyway to get its top-k items. The summaries can be, for example, compact histograms [25] or linear splines [36]. For some applications, we may use even more compact summaries, for example by modelling the (discretized) value distribution of each list as a mixture of two Poisson distributions: $f(x) = \alpha(e^{-\beta}\frac{\beta^x}{x!}) + (1-\alpha)(e^{-\gamma}\frac{\gamma^x}{x!})$. It has been shown [13] that such mixtures are a fairly good approximation of realistic value distributions, especially in text applications.

The first estimation task, i.e., estimating the number of items with a certain minimum score, can be easily performed by lookups in the corresponding list summary. For estimating the number of items with a combined score from a certain subset of lists above a threshold, we first need to compute the convolution of the corresponding summaries to get a summary of the aggregated distribution. This is a cheap computation for histograms and splines; for Poisson mixes, it is even simpler as the convolution of Poisson mixes is again a Poisson mix. We can now estimate the number of items from the convoluted summary.

4.2 Estimating the network costs

After predicting the basic behavior of the algorithms, the optimizer estimates the resulting network costs. The cost is computed using three parameters that can be estimated or determined empirically for a given network:

1. the latency $\mathit{Latency}[i, j]$ between two nodes,
2. the maximum bandwidth $\mathit{Bandwidth}[i, j]$ between two nodes,
3. the effective bandwidth $\mathit{EffBandwidth}[n]$ in a 1:n communication (one node sending messages to its children or all children replying to their parent, in a bursty manner).

```
    networkCosts($O$, $T = \{T_1, \ldots, T_n\}$)
    Input:  a peer $O$; bytes $T_i$ transferred between $O$ and $i$
    Output: the estimated network cost
1   $l = 0; b = 0$
2   for each $T_i \in T$
3       $l = \max(l, 2 * Latency[O, i])$
4       $b = b + \frac{T_i}{\min(Bandwidth[O,i], EffBandwidth[|T|])}$
5   return $l + b$
```

Fig. 2 Cost computation for 1:n transfers

The last parameter models the TCP protocol overhead, as the theoretical bandwidth is usually not achievable. We determined it empirically by running repeated transfer experiments with the ns-2 network simulator; it could be measured the same way in a real system.

Using these parameters, we can estimate the costs of a 1:n transfer, which is the basic network primitive used by the algorithms: one peer requests data from n other peers, first sending the request and then collecting the data. The costs are determined by the highest latency and the total transfer time according to the effectively available bandwidth (see Fig. 2). For entire query trees, the total costs are computed by summing over the slowest path in the tree. In our experiments, the predicted costs were usually within 10% of the real costs.

5 Hierarchical grouping and its optimization

The state-of-the-art algorithms TPUT and KLEE discussed in Sect. 2 employ a flat execution strategy similar to Fig. 1 (a): all queried peers send their data items directly to the query initiator. This execution model is wasteful for a number of reasons. First, it incurs unnecessary communication. Consider, for example, a query with one very large and several small input lists residing on different peers. It would be better to perform the top-k query at the peer with the large list, have the small nodes ship their items to that peer, and only send the final result to the query initiator. Second, the peers compete for network bandwidth, as all of them send their data items to the query initiator forming the top-k aggregation. If instead several peers aggregated data from other peers and only sent their aggregated results to the querying peer, the total bandwidth consumption could be reduced.

We apply a hierarchical grouping of peers in the second phase of these algorithms to reduce transfer costs. Figure 1 (b) illustrates an example execution plan for a query with $m = 4$ input lists L_1 through L_4 on four different peers p_1 through p_4. Instead of querying all four peers for their local items with value above the (uninform) threshold min-$k/4$, the query initiator contacts only peer p_1, which itself contacts p_2 and p_4 with a threshold of min-$k/3$ (the last third of the threshold remains at p_1). Peer p_4 subsequently forwards the request to its children in the execution plan, again dividing the threshold by the respective number of children. For peer p_4, the new threshold is min-$k/(3 * 2)$, as p_4 has two children, including the (local) list L_4. Note that the threshold for the relatively large peer p_2 is higher than the threshold in a flat execution, min-$k/4$, reducing the number of items sent. When p_4 has received all items from its children, it aggregates them with the items of its own list and sends the result to its parent in the execution plan, p_1. As some items may occur in multiple lists, the number of items sent to p_1 is typically less than in a flat execution.

Using such a hierarchical grouping can improve the query execution, but, depending on the sizes and value distributions of the input lists, may also adversely affect performance by adding latency and transfer cost (as data must pass through more than one peer). Therefore, the hierarchical grouping must be constructed by a query optimizer that computes the cost of the candidate trees and chooses the best alternative. The cost of a candidate tree refers to its total execution time, which in our model is dominated by the bandwidth-delimited data transfer times and additional network latencies.

In the following, we first discuss a dynamic programming method for finding the best hierarchical structure and then discuss a fast heuristics to handle larger problems.

5.1 Dynamic programming approach

One way to find the optimal hierarchical structure is to employ dynamic programming (DP) [14]. Note that we only optimize the second phase of the algorithms; so the *min-k* threshold is already known in advance and we only have to organize the aggregation of data items. The cost (i.e., execution time) of each aggregation step is determined by the costs of its slowest input (*max*) and the bandwidth limitations for getting the input data to aggregating peer (basically a weighted *sum*). This allows for the following theorem:

Theorem *The optimal solution of a hierarchical grouping problem can be constructed from optimal solutions for its subproblems* (*i.e.*, *execution trees for subqueries*).

Proof by contradiction. We assume that it is necessary to consider non-optimal solutions of subproblems to find the optimal solution. Then, there exists a problem instance with input lists L such that

1. S is an optimal solution for L and
2. S has a subplan T (i.e., node in the execution tree) that operates on lists $L_T \subset L$ with T being non-optimal for this subproblem and
3. given the cost function c, $c(S) < c(X)$ $\forall$ solutions $X : X$ consists only of optimal solutions for its subproblems

Note that the third condition is implied by the fact that we have to consider non-optimal solutions of subproblems.

Let T' be the equivalent optimal solution for L_T and S' the solution derived from S by replacing T with T'. Obviously $c(T') < c(T)$, as T was non-optimal. W.l.o.g. we assume that S consists of exactly one non-optimal partial solution $\Rightarrow S'$ consists only of optimal partial solutions.

Case 1: T is the root of S. Then $c(T') < c(T) \Rightarrow c(S') < c(S)$. Contradiction to Condition 3.

Case 2: T is a direct child of the root of S, i.e., T is at level 1. Let R be the root node of S. $c(R)$ is determined by the maximum of the costs of its children and a weighted sum for its input sizes. As $c(T') < c(T)$ replacing T with T' will increase neither the maximum nor the weighted sum. $c(R) = c(S) \Rightarrow c(S') \leq c(S)$. Contradiction to Condition 3.

Case 3: T is at a level > 1. Follows by induction from Case 2.

The generalization to more than one non-optimal partial solution can be shown by induction. $\square$

Figure 3 shows the optimization algorithm in pseudo-code; the algorithm applies dynamic programming in a top-down formulation with memorization. The dynamic programming table maps (*lists*, *min-k*) $\rightarrow$ (*peer* $\rightarrow$ *plan*), i.e., for each combination of input lists and *min-k* threshold, we compute and keep the optimal plan for each possible target peer where the subquery result could reside. In our distributed setting, the placement of data also has to be taken into account. This leads to the following optimization process: the algorithm always considers all possible peers as location for the result, i.e., it operates on sets of plans—one plan for each possible peer where the final result could reside. A (sub-)problem can always be solved by using a flat execution, i.e., aggregating the input peers at the target (lines 4–5). If the problem consists of more than one input peer, the aggregation can instead be performed hierarchically: the problem is split into smaller problems whose results are then combined (lines 6–

```
     buildHierarchy($L$,min-k)
     Input:   set $L$ of all data-item lists; value threshold min-k
     Output:  set of optimal execution plans, one for each peer
1    if ($I$, min-k) has already been solved
2        return known solution
3    $b$ = empty plan set
4    for each $p \in$ peers
5        $b[p]$ = flat aggregation of $L$ at $p$, threshold min-k
6    if $|L| > 1$
7        for each $P = \{L_i \subset L\}$, $P$ partitioning of $L$
8            $L' = \{\text{buildHierarchy}(L_i, \textit{min-k}/|P|) | L_i \in P\}$
9            for each $p \in$ peers
10               $L_p = \{i[p] | i \in L'\}$
11               $a$ =aggregation of $L_p$ at $p$
12               if $a$.costs< $b[p]$.costs
13                   $b[p] = a$
14       for each $p_1, p_2 \in$ peers
15           if transfer($b[p_1]$, $p_2$).costs< $b[p_2]$.costs
16               $b[p_2]$ =transfer($b[p_1]$, $p_2$)
17   store $b$ as solution for ($L$, min-k) in DP table
18   return $b$
```

Fig. 3 DP algorithm for optimal grouping

13). As it might be better to perform the entire aggregation at one peer and merely ship the results, the algorithm considers the cost of this case (lines 14–16).

To assess the quality of an execution tree, the algorithm estimates its transfer cost. For the transfer cost, the number of items transferred from a group of peers to their parent is estimated using the statistical prediction model of Sect. 4.1. For large m, we can use a faster heuristics instead of the exact DP (see below) and/or use sampling (see Sect. 7). It is difficult to determine tight bounds for the algorithm complexity, as search space pruning depends on the concrete problem instance. Note that the pseudo-code is simplified, it shows the search space organization but hides several implementation details. The DP algorithm can be implemented with an upper bound of $O(m4^m)$. Unfortunately $\Omega(2^m)$ is a lower bound which makes using DP infeasible for large m.

5.2 Fast heuristics

Dynamic Programming finds the optimal hierarchical structure, but its run-time may be prohibitively high, as trying out all partitionings (line 7) becomes infeasible when the number of lists to aggregate is too large. To avoid the exhaustive search, we use the following fast heuristics to find a good partitioning. The hierarchical structure is basically a divide-and-conquer strategy for the aggregation; therefore, we want to partition the lists such that the resulting partitions exhibit approximately equal costs. In our cost model, lists with similar cardinality will cause similar effort; so we heuristically partition the list L of all data-item lists as follows:

- S_L: L sorted by cardinality of items above $min\text{-}k/|L|$, i.e., number of "relevant" items in each list
- O_L: every "odd" list of S_L $(L_1, L_3, \dots)$ (sorted by desc. cardinality)
- E_L: every "even" list of S_L $(L_2, L_4, \dots)$ (sorted by asc. cardinality).

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

6 Adaptive thresholds

After determining an initial *min-k* threshold, both TPUT and KLEE in their second phase request all data items that may possibly qualify for the top-k results. TPUT takes a conservative approach by distributing the necessary value mass uniformly over all input lists and requests all items with a local value above $min\text{-}k/m$ (cf. Fig. 1 (a)). However, as value distributions vary widely across lists, data-adaptive thresholds that are specifically tuned to the individual lists (cf. Fig. 1 (c)) are promising and were already considered in [44], but deemed computationally intractable and not pursued much further. Our approach chooses adaptive thresholds by first choosing scan depths (number of items to be transferred) and then deriving appropriate value thresholds $min\text{-}k/m$.

We can formally define this optimization problem as follows. Assume that we scan the m input lists to depths $d_1, d_2, \ldots, d_m$, and the values at these list positions are $v(d_1), v(d_2), \ldots, v(d_m)$, respectively. We need to ensure that we scan deep enough so as not to miss any potential top-k candidate; this mandates the constraint $\sum_{i=1}^{m} v(d_i) \le min\text{-}k$, with uniform thresholding being a special case. We aim to minimize the total cost of shipping list entries, which is equivalent to minimizing $\sum_{i=1}^{m} d_i$, subject to the introduced constraint. For given scan depths d_i we can estimate the resulting $v(d_i)$ by using our probabilistic predictors developed in Sect. 4.1. This problem is NP-hard, as we can reduce the Knapsack problem to our problem:

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

Given an instance of KNAPSACK, we construct the following instance of the threshold-adaption problem as follows. We consider m lists where the ith list l_i consists of a single entry with score u_i. The cost to read an entry from list i is w_i. This trivial transformation can obviously be done in polynomial time. Choosing an item X_i in the traditional KNAPSACK terminology corresponds to reading an entry of list l_i.

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

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

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

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

The key idea is to optimize the maximum scan depth over the m lists (instead of the sum of the scan depths). In a lightly loaded network with all m scans proceeding in parallel on different peers, this objective function would be appropriate for minimizing the latency of this phase. For our actual objective function, minimizing the total network costs (Sect. 4.2), it is merely a heuristics, but turns out to be a fairly good approximation in practical settings. If we minimize the deepest scan, i.e., $\max_{i=1}^{m} d_i$, we can set all d_i to the same maximum, so that we effectively deal with only one free variable as $d_1 = d_2 = \cdots = d_m$. We still need to ensure that this choice of d_i satisfies the constraint. But now we can easily perform a binary search over the possible choices, to find the lowest d_i without violating the constraint. Note that this approach of uniform scan depths usually results in non-uniform local thresholds at which the scans on the individual lists stop. Further note that each step of the binary search requires evaluating our single-list cost prediction model for each peer. Here we use the model based on linear splines (rather than Poisson mixes) as it is crucial to capture the specific distributions of individual lists and to do so with high accuracy. In our implementation, the overhead of these computations is negligible.

7 Site sampling

For distributed queries that involve hundreds of peers, interactive response times can hardly be achieved even with all the optimizations described earlier. For such cases, we additionally consider an approach based on sampling that operates only on a small fraction of randomly chosen input lists.

We introduce the following two sampling dimensions:

1. Instead of considering all n items per list, consider only the top n' items of each list.
2. Instead of considering all m lists, consider only a sample of m' lists.

For the second option, the sample may be chosen in either a uniform or in a data-adaptive manner. Here we focus on the adaptive selection of m' lists of those peers p_i with the highest value masses $w_i := \sum_{j=1}^{n} v_i(I_j)$ over all their local items I_j. We assume that we know the fraction of the total value sum that these m' lists accumulate (e.g., by means of per-peer histograms or other synopses):

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

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

We denote by $\textit{min-k}(m, n)$ the final *min-k* value of any exact non-sampling algorithm, and by $\textit{min-k}(m', n')$ the final *min-k* value with our sampling-based algorithm that considers the top n' entries from m' lists. The linear error $|\textit{min-k}(m, n) - \textit{min-k}(m', n')|$ is a measure of the accuracy of the sampling-based top-k algorithm, and we can use this error measure for calibrating the choices of m' and n'. However, during query execution we do not know $\textit{min-k}(m, n)$, so we first need to estimate this value using per-peer summaries as defined in Sect. 4.1.

7.1 Estimating $\textit{min-k}(m, n)$

We consider a top-k query over input lists L_i, $i = 1, \ldots, m$, spread across m peers, and want to predict the value of *min-k*, i.e., the aggregated value of the rank-k item in the global ranking. For the sake of tractability, we assume that all index lists have the same length n, and that all items occur in every list (but possibly with a score of zero), i.e., there are n distinct items. We additionally assume stochastic independence between the different lists. Although the latter assumption rarely holds in practice, models based on independence have been very successful for many prediction tasks and applications that require such statistical reasoning.

Now let the top-k algorithm read all entries of all lists for simplicity (any pruning performed by the algorithm will not affect the top-k items anyway). Under our assumptions, all items will be seen in all lists. For each item we want to characterize its total value that results from the aggregation over the m lists. We denote these aggregated values by the random variable S. The distribution of S, $f_S(x)$, is obtained by the convolution of $f_{Si}(x)$, the distribution in each list.

Each of the n items that we see has an aggregated value according to the probability density function $f_S(x)$. Denote these random variables by $T_1, \dots, T_n$ and order them in ascending order. Without loss of generality, we can renumber them such that $T_1 \leq T_2 \leq \cdots \leq T_n$. We are interested in the value of the rank-k item, namely T_{n-k+1}. This estimation problem is a standard problem in order statistics [17]. T_{n-k+1}, the rank-k order statistics, is itself a random variable, which is difficult to characterize in its full distribution. But we are only interested in its expectation $E[T_{n-k+1}]$. A first-order approximation to this is the $((n-k+1)/n)$-quantile of $F_S(x)$; more accurate approximations based on a Taylor-series expansion can be derived [17] but are difficult to compute (including evaluating derivatives of the quantile function). We will therefore use the simpler first-order approximation

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

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

If we represent S with a histogram, the quantile can be efficiently calculated done by binary search on the histogram cells.

7.2 Estimating *min-k*(m', n')

Analogously to the prediction model in the previous subsection, we can estimate the sampling-based min-k value when considering only m' lists, but all items in each list. We only have to replace the convolution of all m lists by the convolution of the m' lists used during execution.

The estimation is more difficult (and at the same time less accurate) when sampling only n' items from each list, considering a subset of m' lists (the items are typically non-disjoint across lists). As we will typically not see every item in every list, we need to estimate the expected number of lists, m'', in which we see an item, and the expected number of distinct items, n'', seen in all lists. We make the conservative error of assuming that the n' items are uniformly drawn among the items in a list (whereas in reality we draw the top-n' items).

With uniformly chosen m' lists:

$$P[\text{item seen in } q \text{ out of } m' \text{ lists}]$$
$$= p_{seen}(q) = \binom{m'}{q}\left(\frac{n'}{n}\right)^q \left(\frac{n-n'}{n}\right)^{m'-q}$$

with expectation $E_{seen} = m'n'/n := m''$. The probability that we see item d in at least one list is $1 - P_{seen}(0) = 1 - \frac{n-n'}{n}^{m'} = 1 - (1 - \frac{n'}{n})^{m'}$, and the expected number of distinct items seen in all lists together is $E_{dist} = (1 - (1 - \frac{n'}{n})^{m'})n$. Now we make the simplifying assumption that each of the E_{dist} items is seen in exactly m'' lists. As we don't know in which lists the item will occur, we cannot compute the convolution of the score distribution in those lists. Instead, we need to assume that all lists have a similar score distribution that can be modeled, for example, as Poisson mix. Now we can estimate the convolution of the m'' lists by computing the m''-fold convolution of this distribution and estimate *min-k*(m', n') as shown in the previous subsection.

With the non-uniform sampling strategy that selects the m' "heaviest" lists, the analysis of E_{seen} and E_{dist} is analogous. To take into account the fact that some peers have much "heavier" value mass than others, we now consider also the peer-specific w_i values and adjust the Poisson-mixture parameters in the estimation of *min-k*(m', n') as follows. We assume that all m' lists have the same value distributions but together constitute fraction φ of the overall value sum over all lists. Thus, the expected value of an item in one of the m' lists is larger than the expected value in a model with all lists having equal weight by the factor $\varphi m/m' =: \rho$, the "boost factor". We then adjust the parameters of the per-list Poisson-mixture model to have this boosted expectation. The expectation of the non-weighted Poisson mix is $\alpha\beta + (1-\alpha)\gamma$. This easiest way of boosting the expectation then is by setting $\beta' := \rho\beta$ and $\gamma' := \rho\gamma$, yielding the expectation $\rho(\alpha\beta + (1-\alpha)\gamma)$ of Sect. 4 for estimating the adjusted *min-k*(m', n').

This way the selection of the peers to be sampled can be made locally by the query initiator, without contacting other peers. All it requires is global knowledge of the Poisson-mix parameters α, β, γ, and the peer-specific masses w_i. As these values change infrequently, we can periodically re-estimate them and disseminate them to all peers.

8 Node failures and network dynamics

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

When the query originator fails, the query can be aborted anyway; if the failure is transient, the query originator can resubmit the query after its restart. When a node fails that was involved in message routing (e.g., an intermediate node in DHT-based routing) but is not involved in the query execution, we employ whatever routing redundancy the underlying network provides (there is ample literature on dynamic rerouting in the networking and P2P systems community).

The remaining, not so straightforward, case is when one of the peers fails that is involved in the query execution tree. When a peer realizes that its parent has failed, the techniques of [1] can be applied: the orphaned peer sends its results either directly to the query initiator or to some known ancestor in its caller tree. Conversely, when a peer realizes that one of its children has failed, it may either find alternative routes to reach its affected grandchildren or it could view the entire subtree as unavailable. Such steps may even include dynamic re-optimizations, e.g., to adjust thresholds. Exploring approaches along these lines is left for future work.

9 Experiments

9.1 Setup

We have implemented all algorithms and our testbed in C++. The distributed nature of the algorithms complicates experiments, in particular since we want to measure effects of (sometimes small) variations to execution strategies. Experiments in real-world networks like PlantetLab, while highly desirable, add a significant amount of noise and are hard to control for systematic experiments. To obtain reproducible and comparable results, we simulate the network. As a simulation environment we used the ns-2 network simulator [24], which is a highly detailed and validated, state-of-the-art packet level network simulator. It mimics the 4.x BSD TCP stack and thus accurately models the behavior of real systems. We used the Inet-3.0 topology generator [42] to create an Internet-style topology with 3037 nodes and bandwidths ranging from 1 MBit/s in the leaves to 10 GBit/s in the backbone. For the experiments, we assigned the individual data lists to random nodes, and employed the network simulator to compute the execution time for each algorithm given the network topology and the data placement. The run-times include the complete network traffic, including opening connections, SYN and ACK packets, TCP send window effects, etc. All results are averages over 10 random placements.

Algorithms under comparison

We have presented three different optimizations. In principle any combination of these techniques is possible, but our experimental evaluation focuses on the following instances:

TPUT is the three-phase uniform threshold algorithm [9]. We do not consider the variant of TPUT that uses a compression technique based on hash array encoding to decrease the network bandwidth consumption. We consider it an orthogonal issue to apply compression techniques to any of the investigated algorithms.

KLEE is an extension of TPUT that employs histograms and Bloom filters to increase the $min\text{-}k/m$ threshold [33]. It is an approximate algorithm, i.e., it does not guarantee to find the exact top-k query results. The overall performance of KLEE depends on a parameter c that determines the number of Bloom filters that are transferred in the first execution phase as a fraction of the total value mass of an input list. We set $c = 5\%$. We use only the three-phase KLEE variant, and disregard the KLEE-4 variant of [33] as its additional filtering step would be orthogonal to the issues studied here.

AdaptiveTPUT is an extension of TPUT that uses our adaptive thresholding described in Sect. 6, i.e., the second execution phase is enhanced by non-uniform thresholds. AdaptiveKLEE is the equivalent extension of KLEE.

TreeTPUT uses hierarchical query execution plans resulting from the optimizations introduced in Sect. 5, in addition to the adaptive-threshold technique of AdaptiveTPUT. Adding hierarchical execution plans to AdaptiveTPUT adds the flexibility to solve a problem by either using a flat aggregation or by splitting it into smaller problems and solving them hierarchically (while still also using adaptive thresholding). In the experiments, we use the dynamic-programming algorithm for m up to 10 and switch to the fast heuristics for higher m. In general the threshold should be chosen such that the expected gain is larger than the expected run-time. Both can be estimated reasonably, we used a fixed threshold to avoid CPU dependency. TreeKLEE is the equivalent extension of KLEE.

SamplingTreeKLEE additionally extends TreeKLEE by utilizing the sampling techniques described in Sect. 7; i.e., SamplingTreeKLEE combines all three optimizations presented in this paper. We sample a number of peers, in descending order of value mass, so that our *min-k* estimate predicts a maximum error of at most 20% compared to the *min-k* estimate if we ran the query on all peers. In the experiments this resulted in typically selecting between 10 and 30 percent of the peers involved in a query. We did not combine sampling with TPUT, as in contrast to KLEE TPUT is designed as an exact algorithm. If desired one could combine sampling with the approximate variant of TPUT.

Approximate vs. exact mode

KLEE has explicitly been designed as an approximate algorithm [33]. However, KLEE and all non-sampling methods can be turned into exact algorithms by adding an additional random-lookup phase at the end. The resulting algorithms can be considered as TPUT variants flavored with KLEE's techniques plus our optimization techniques. On the other hand TPUT has been designed as an exact algorithm [9], but can be transformed into an approximate algorithm by skipping the random-lookup phase at the end. In our experiments we study both TPUT and KLEE both in exact and in approximate mode.

Datasets

The WorldCup HTTP server log collection[1] consists of about 1.3 billion HTTP requests recorded during the 1998 FIFA soccer world cup. The data was served by 33 load-balancing web servers distributed over Europe and the US, and is provided as

[1] http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

249 individual access logs. We constructed 249 peers by converting each access log into a peer. The task is to identify the top-100 clients that caused the most traffic on a given set of peers.

AOL Query Log:[2] This search-engine query log consists of ∼20M queries collected from ∼650k users over three months. We have considered all `(userid, terms, date)` triplets (userid provides a stable mapping from queries to users over the entire time period). We have grouped the queries by userid and, for each user, created all possible term pairs from her queries after applying stemming and stopword elimination. We have finally created 5,000 peers from the users with the highest numbers of different term pairs. Here, a top-k query consists of a set of m users and the task to find the top-100 most frequent term pairs that occur in the queries issued by the users over the complete time interval.

The Retail Benchmark consists of retail market basket data from an anonymous Belgian retail store [6]. A set of 100 peers was generated by randomly assigning each of the ∼88k transactions to exactly one peer, modeling a situation in which the transactions had occurred at distributed shopping sites. At each peer, we generated all possible triplets of basket items present in any of the transactions, yielding a total number of 51,788,094 (16,769,821 distinct) triplets. As for queries, we are interested in finding the globally most frequent triplets, using only a subset of the 100 peers (i.e., retail stores).

Performance metrics

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

Cost factor query response time: Elapsed "wall-clock" time for the benchmarks, using the network simulation (see above) for deriving elapsed time.

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

9.2 Results

Figure 4 (left) shows the average query response times for the Retail benchmark for different query sizes (number of peers queried), when all algorithms operate in exact mode. Each point in the chart is computed by averaging over 10 independently chosen random queries for the given number of peers (i.e., appropriately chosen query parameters). For all queries, TPUT is improved by AdaptiveTPUT and further improved by TreeTPUT. Interestingly, KLEE performs worse than TPUT, and AdaptiveKLEE performs slightly worse than KLEE for query size 20. This is caused by additional random lookups in Phase 3: while KLEE indeed retrieves less data items in Phase 2, it requires more random lookups in Phase 3, which are quite expensive. For query size 20, the scan depth balancing aggravates this by reading even less data items and

[2] http://www.gregsadetsky.com/aol-data/.

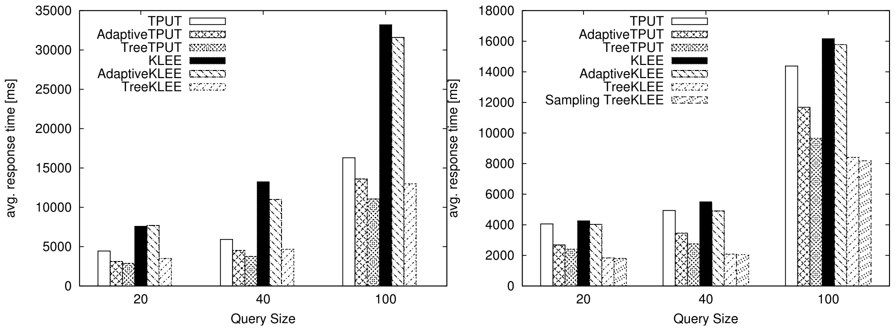

Fig. 4 Retail results in exact (*left*) and approximate (*right*) mode

thus requiring more random lookups. For larger query sizes 40 and 100 the benefit of adaptive scan depths outweights the additional random lookups. The TreeKLEE variant performs much better than KLEE and AdaptiveKLEE, but is still slower than TreeTPUT.

Figure 4 (right) illustrates the average query response times when all algorithms operate in approximate mode; Table 1 shows the relative recall for the largest queries. The differences between TPUT and KLEE are smaller here, but still KLEE is slower due to its relative expensive Phase 1 communication (which does not pay off here). Only the TreeKLEE optimization can make use of the improved thresholds gained from Phase 1 and thus performs better than TreeTPUT. Overall AdaptiveKLEE is only a minor improvement over KLEE here, while TreeKLEE improves the run-time of KLEE up to a factor of two. For TPUT the adaptive scan depths have a much larger impact, with AdaptiveTPUT performing nearly as good as TreeTPUT for small queries.

Figure 5 (left) shows the average query response times for the AOL benchmark where all algorithms operate in exact mode. Both TPUT and KLEE perform similar here, with AdaptiveTPUT/AdaptiveKLEE improving the performance up to 10% and TreeTPUT/TreeKLEE improving the run-time up to a factor of two. It is noticeable that the optimization gains go down with increasing query size. For 300 peers the Adaptive variants perform nearly the same as the unoptimized versions, and the Tree variants reduce the run-time by approx. 24%. The explanation for this is the extremely massive network load for the large queries. The queries with 300 peers involve nearly 10% of the whole network, which causes a massive overhead due to TCP/IP messaging. The volume of the network traffic itself is not that large (ca. 5 MB on average) but the network simulator showed that the data cannot be sent efficiently anymore due to the bursty traffic between a large number of interacting peers.

When the algorithms operate in approximate mode (Fig. 5 (right)), the results for TPUT and KLEE are similar to the exact mode. However SampleTreeKLEE performs much better, improving the run-time of KLEE by more than a factor of 5, while maintaining a relative recall of 90% (cf. Table 1). This indicates that for very large queries sampling is a must, as otherwise the network itself cannot handle the query load.

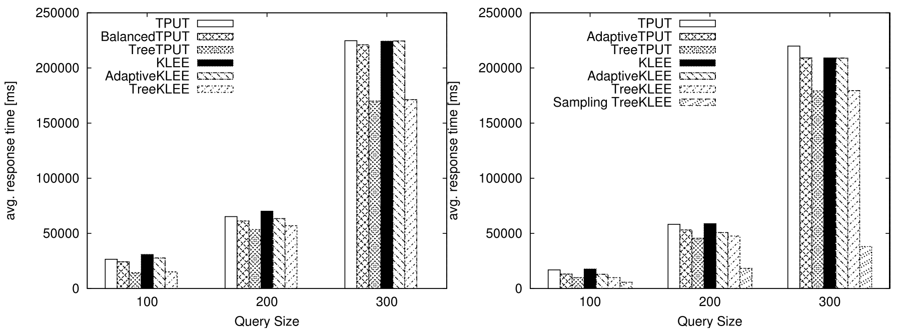

Fig. 5 AOL results in exact (*left*) and approximate (*right*) mode

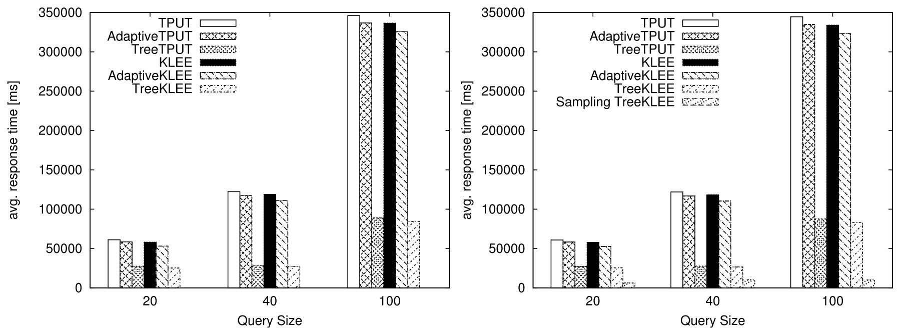

Fig. 6 Worldcup results in exact (*left*) and approximate (*right*) mode

Figure 6 illustrates the average query response times for the WorldCup benchmark. With all algorithms operating in exact mode (shown in Fig. 6 (left)) AdaptiveTPUT/AdaptiveKLEE improve the average response time only slightly over TPUT and KLEE, whereas TreeTPUT and TreeKLEE improve the run-time up to a factor of four. Overall TreeTPUT and TreeKLEE performed similar, which suggests that the effect of the hierarchical optimization dominates the differences between TPUT and KLEE here.

Table 1 shows the relative recall numbers for different numbers of peers, for all algorithms operating in approximate mode. The run-time effects of the optimization algorithms for TPUT and KLEE (Fig. 6 (right)) are similar to the exact case. In this scenario sampling has very low recall values, which is caused by the data distribution of the Worldcup data set: the top entries are randomly distributed among the peers, which means that any strategy that considers only a subset of peers must lose recall. Thus, sampling is no panacea; it assumes that there is some exploitable correlation in the data, which is often the case, but not always. Still the non-sampling TreeKLEE strategy performs very well, and can be used when sampling is not appropriate.

Table 1 Recall results in approximate mode

#Peers	TPUT	Adaptive TPUT	Tree TPUT	KLEE	Adaptive KLEE	Tree KLEE	Sampling TreeKLEE
				Retail			
40	0.98	0.97	0.97	0.92	0.90	0.90	0.85
100	0.98	0.96	0.96	0.92	0.90	0.90	0.85
				AOL			
200	0.99	0.98	0.98	0.99	0.98	0.98	0.90
300	0.99	0.97	0.97	0.98	0.97	0.97	0.89
				Worldcup			
40	0.98	0.96	0.95	0.97	0.95	0.94	0.22
100	0.99	0.97	0.96	0.98	0.96	0.95	0.13

9.3 Prediction accuracy

The optimization methods rely on the predictions discussed in Sect. 4 for *min-k* values and execution costs. To study the accuracy of these estimations, we compare the distribution function induced by the actual data with the distribution function implied by the histograms. Notice that, by comparing the distribution functions directly, we avoid depending on the actual values of k.

The base histograms over unaggregated data have an average relative error of $< 0.5\%$ for all lists in all data sets. So the estimations of the original, unaggregated data (used by our non-hierarchical AdaptiveTPUT/AdaptiveKLEE) are nearly perfect, the estimation errors are negligible. When aggregating data we cannot expect this level of accuracy, as we have to combine already aggregated data and assume independence in the convolutions. Still, in experiments on the AOL data set, the error remains $< 14\%$ even for 10 convolutions and more. The error is probably caused by correlations between the lists: While the Worldcup dataset behaves similar to the AOL dataset, the estimations performed really well on the Retail dataset: the accuracy after a single convolution is very good ($< 2\%$), and remains roughly at this level even for 10 convolutions and more.

9.4 Discussion

Overall, TreeTPUT/TreeKLEE are the best-performing and most robust algorithms. They are superior to all other competitors in all cases, and significantly outperform the base algorithms with run-time gains up to a factor of four. In exact mode, TreeTPUT is slightly preferable to TreeKLEE (which is not surprising, as KLEE was designed as an approximate algorithm). In approximate mode, TreeKLEE performs better than TreeTPUT and is the algorithm of choice. The optimizations themselves have a greater impact than the choice of the base algorithm: while TPUT and KLEE themselves perform quite differently, TreeTPUT and TreeKLEE are much closer to each other. Optimizing only the scan depths in AdaptiveTPUT/AdaptiveKLEE already improves the run-times, but the full cost-based optimizations of TreeTPUT and TreeKLEE give much better results and are essential for consistently good performance.

SamplingTreeKLEE has even shorter response times than TreeKLEE, but the two methods are actually incomparable as SamplingTreeKLEE is inherently approximate and typically exhibits a non-negligible loss in relative recall. Notwithstanding this observation, SamplingTreeKLEE is the method of choice for very high m and it achieves a very impressive quality/cost ratio. For queries with 300 peers, SamplingTreeKLEE outperforms all other methods, including TreeKLEE, by a factor of five while still retaining decent result quality with relative recall often above 90 percent.

Although the issue of exact vs. approximate results is orthogonal to the contributions of this paper, we think it is worthwhile pointing out that the approximate variant of TreeKLEE is a particularly intriguing algorithm for many practical applications. It is often a factor of two faster than its exact counterpart, but consistently achieves a relative recall above 90 percent or higher—an excellent result quality that would be perfectly acceptable for most applications of top-k querying.

10 Conclusion and future work

This paper has developed and experimentally studied novel techniques for optimizing top-k aggregation queries that involve many peers in a wide-area network. Each of our main techniques can individually improve the performance of the state-of-the-art algorithms, TPUT and KLEE. Together, our techniques exhibit additional synergies and consistently outperform prior methods. We believe that distributed top-k querying will gain even more practical importance with the further proliferation of network-centric applications, such as network monitoring or mining of social communities. Our future work will aim to eliminate the few limitations that our methods have: i) generalizing beyond the current restriction to monotonic aggregation functions (e.g., supporting top-k average or median), ii) considering correlation information for the underlying peers and their value distributions in the statistical predictor models, and iii) looking for better approximation techniques for our hierarchical grouping and adaptive thresholding methods, with the goal of being scalable to very large m while yielding near-optimal plans.