Efficient Processing of Distributed Top-k Queries

Hailing Yu Hua-Gang Li Ping Wu Divyakant Agrawal Amr El Abbadi

Department of Computer Science
University of California, Santa Barbara, 93106, USA
{hailing,huagang,pingwu,agrawal,amr}@cs.ucsb.edu

Abstract

Ranking-aware queries, or top-k queries, have received much attention recently in various contexts such as web, multimedia retrieval, relational databases, and distributed systems. Top-k queries play a critical role in many decision-making related activities such as, identifying interesting objects, network monitoring, load balancing, etc. In this paper, we study the ranking aggregation problem in distributed systems. Prior research addressing this problem did not take data distributions into account, simply assuming the uniform data distribution among nodes, which is not realistic for real data sets and is, in general, inefficient. In this paper, we propose three efficient algorithms that consider data distributions in different ways. Our extensive experiments demonstrate the advantages of our approaches in terms of bandwidth consumption.

1. Introduction

Ranking-aware queries have been studied in various contexts such as web, multimedia retrieval, relational databases, and distributed systems. This is mainly because they are needed for decision-making related activities such as identifying interesting objects, network monitoring, distributed denial-of-service attack detection and load balancing. For example, in a network monitoring setting, top ranking sources of data packets need to be identified to detect denial-of-service attacks. Alternatively, consider an online advertisement tracking scenario, where each advertiser places its advertisements on different publishers' pages, e.g., CNN and BBC. In such an application, advertisers will often be interested in identifying the top publishers that generate advertisement driven revenue. Fagin first introduced the ranking aggregation problem in the context of multimedia retrieval [6]. Assume there are m subsystems with each maintaining a list of objects together with their ranking scores. The score of an object can be any value that describes a certain characteristic of the object, e.g., its color, shape, etc. Top-k queries over the m subsystems return the objects with the k highest aggregated scores under a monotonic function. The best known algorithm solving this problem is the *Threshold Algorithm* (TA) which was independently discovered by several groups [8, 11, 7]. Based on Fagin's seminal work [6], many approaches have been proposed to solve the top-k query processing problem under various scenarios. In data streams, distributed top-k monitoring was studied in [1]. Supporting ranking query processing in relational databases from different perspectives has been studied in [3, 5, 9, 10, 15, 14]. More recently, approximate top-k queries on multidimensional datasets with probabilistic guarantees were studied in [13]. A framework for distributed top-k retrieval in peer-to-peer networks was proposed in [2], which is mainly concerned with retrieving the top-k matching objects given the query object, but does not aggregate scores from all nodes in a distributed system.

In this paper, we are concerned with answering top-k queries efficiently in distributed systems. In particular, we consider *Content Distribution Networks* (CDNs), which are deployed by many companies to avoid network congestion. CDNs typically consist of *cache servers* scattered around the globe for caching bandwidth-intensive objects from the *original server* such as images and video clips. This enables fast web and streaming media applications. When a request is sent to the original server, it is redirected to one of the cache servers which is closer to the client and/or can serve data faster. Effective monitoring of activities (by a *central manager*) over CDNs ensures successful content distribution. One such monitoring task is a top-k query, e.g., "*what are the top-k most popular URLs across the entire CDN?*". A naïve approach to answer such a query is to have each cache server send the access statistics about all objects to the central manager. However, this incurs significant bandwidth consumption if the number of objects at each cache server is large. Hence bandwidth efficient algorithms for processing such top-k queries in a distributed environment are needed.

While the Threshold Algorithm (TA) is generally applicable in database applications, it is inefficient when applied to answer top-k queries in large distributed networks in terms of bandwidth consumption [4]. This is mainly because the number of rounds to finalize the answer to a top-k query under TA cannot be predetermined and it varies with different data distributions among the nodes. Hence, in [4] the first constant number of round algorithm for calculating top-k objects in distributed systems is proposed and referred to as the *Three-Phase Uniform-Threshold* algorithm (TPUT).

However, TPUT does not take data distributions into account and it simply assumes the uniform data distribution among all nodes, which is not realistic due to the heterogeneous nature of distributed systems. Thus, in this paper, we propose different algorithms to calculate top-k queries in constant number of rounds to further enhance the performance by accounting for varying data distributions. They are referred to as the *Three-Phase Adaptive-Threshold* algorithm (TPAT), the *Three-Phase Object-Ranking* based algorithm (TPOR) and the *Hybrid-Threshold* algorithm (HT). TPAT generalizes TPUT by utilizing summary statistics of the data. However, it could be very expensive to use summary statistics to accurately estimate data distributions. The main difficulty is for an algorithm to efficiently estimate data distributions, without *a-priori* knowledge. TPOR and HT are devised to overcome this difficulty. TPOR is fundamentally different from both TPUT and TPAT since it uses object rankings rather than object scores when estimating data distributions. TPOR is more bandwidth-efficient than TPUT when handling the case that object rankings are similar across all nodes. Nevertheless, TPOR performs worse than TPUT in the case when object rankings widely vary across all nodes. To remedy such a situation, HT is proposed to combine the advantages of both TPUT and TPOR, which is robust under different data distributions.

The rest of the paper is organized as follows. Section 2 formulates the problem. Section 3 reviews TA and TPUT to summarize the state of the art. Section 4 presents our proposed algorithms. Section 5 evaluates the performance of our proposed algorithms. Section 6 concludes the paper and discusses the future work.

2. Problem Formulation and Performance Metric

We formalize the problem of top-k query processing in distributed systems by abstracting the above CDN example. Assume there are m nodes and one single *central manager* in a distributed system. Each node i is connected to the central manager and maintains a list of pairs $\langle O, S_i(O)\rangle$, where O is an object and $S_i(O)$ is the score of the object. Furthermore, we assume objects in each list are sorted in the descending order of their scores. Note that an object does not have to appear in all nodes. If an object does not appear in the list of a node, its score in that list is zero by default. The central manager initiates a top-k query which retrieves objects from the network with the k highest $f(S_1(O), \ldots, S_m(O))$ where f is a monotonic function such as the sum function `SUM` to compute the overall score of an object. For simplicity, we assume the sum function throughout this paper. In practice, this function could be a weighted sum to account for the relative importance of cache servers.

The goal of distributed top-k query algorithms is to achieve low bandwidth consumption. We assume that the computation cost in each node is negligible while the communication cost among nodes dominates the query response time. This is mainly due to the current trends in technology where the speed and bandwidth of the network is still a bottleneck. In this paper, we take the number of $\langle object, score\rangle$ pairs transmitted across the network as our performance metric, which dominates the communication cost.

3. Background

This section introduces some background of the ranking aggregation algorithms. In particular, the *Threshold Algorithm* (TA) and the *Three-Phase Uniform- Threshold* (TPUT) algorithm are discussed in detail.

3.1. Threshold Algorithm

Fagin [6] introduced the rank aggregation problem for multimedia database systems. Recall that there are m subsystems each maintaining a list of objects with their scores. Each list is sorted based on the object scores. There is a predefined monotonic function f which returns an aggregated score for a given object. The Threshold Algorithm (TA) is an improvement of Fagin's Algorithm (FA) in [6]. TA scans the lists in the subsystems simultaneously from the top most ranked objects. In each round, only one object is retrieved from each list. TA maintains a *threshold* value that gives the upper-bound of the aggregated scores of all objects not yet seen. Assume the last objects seen from all the lists are $O_{last}(i)(1 \leq i \leq m)$ and their respective scores are $S_i(O_{last}(i))$. The threshold value is calculated as $f(S_1(O_{last}(1)), \ldots, S_m(O_{last}(m)))$. This threshold value is recomputed every time TA sees new objects from the m lists. When a new object O_{new} is seen, TA retrieves its scores by random accesses from those subsystems where O_{new} has not been scanned yet. Hence, the actual aggregated scores of the objects seen so far can be calculated. TA maintains at most k objects at any time. Once TA finds k

objects whose aggregated scores are no less than the current threshold value, it terminates and returns them as the top-k objects.

	subsystem 1	subsystem 2	subsystem 3
1	<O_5, 21>	<O_4, 34>	<O_3, 30>
2	<O_2, 17>	<O_1, 29>	<O_4, 14>
3	<O_4, 11>	<O_0, 29>	<O_0, 9>
4	<O_3, 11>	<O_3, 26>	<O_5, 7>
5	<O_6, 10>	<O_5, 9>	<O_2, 1>
6	<O_7, 10>	<O_9, 7>	<O_8, 1>

Figure 1. An example with 3 subsystems

Example 1 *Figure 1 shows an example with three subsystems. Assume that the top-k query requests the top 2 objects and the monotonic function f is the summation function* SUM*. TA first scans the top most objects in all subsystems which are O_5, O_4, and O_3. Hence the threshold value at this time is $21 + 34 + 30 = 85$. Then TA calculates the aggregated score for each object seen so far by random accesses to the three subsystems. We get the aggregated score for O_5, $S_{\texttt{sum}}(O_5) = 21 + 9 + 7 = 37$, for O_4, $S_{\texttt{sum}}(O_4) = 11 + 34 + 14 = 59$, and for O_3, $S_{\texttt{sum}}(O_3) = 11 + 26 + 30 = 67$. TA maintains the top 2 objects seen so far, which are O_3 and O_4. As neither of them has an aggregated score greater than the current threshold value, TA continues to scan the objects at the second positions of all lists. At this time, the threshold value is recomputed as $17 + 29 + 14 = 60$. The new objects seen are O_1 and O_2. Their aggregated scores are retrieved and calculated as $S_{\texttt{sum}}(O_1) = 0 + 29 + 0 = 29$, and $S_{\texttt{sum}}(O_2) = 17 + 0 + 1 = 18$. TA still keeps objects O_3 and O_4 since their aggregated scores are higher than those of both O_1 and O_2. Since only O_3 has an aggregated score greater than the current threshold value, TA algorithm continues to scan the objects in the third positions. Now the threshold value is $11+29+9 = 49$ and the new object seen is O_0. TA computes the aggregated score for O_0, which is 38. O_3 and O_4 still maintain the two highest aggregated scores, which are now greater than or equal to the current threshold value. Thereby, TA terminates at this point and returns O_3 and O_4 as the top 2 objects.*

When adapting TA directly to distributed systems, a lot of communication overhead is incurred since TA has to go round by round and each round involves two round-trip communications [4]. In the first round-trip, the central manager asks each node in the network to send one object at a certain position. In the second round-trip the central manager sends a list of object IDs to every node in the network and asks them to send the scores of these objects. As the number of rounds in TA is arbitrary (dependent on the data distribution), this results in unpredictable performance, which is not desirable in distributed systems. As an alternative to reduce the number of rounds, batching objects in the first round-trip does not help much as shown in [4]. Hence, new algorithms are needed that can terminate deterministically. Such an algorithm is proposed in [4], which is referred to as the *Three-Phase Uniform-Threshold Algorithm*, and requires three rounds to return the exact top-k objects.

3.2. Three-Phase Uniform-Threshold Algorithm

To describe the Three-Phase Uniform-Threshold Algorithm (TPUT), we first introduce the notion of *partial sums* of objects which are calculated by the central manager. For an object O, the partial sum $S_{\texttt{psum}}(O) = S'_1(O) + \ldots + S'_m(0)$ where $S'_i(O) = S_i(O)$ if O has been reported by node i to the central manager, and $S'_i(O) = 0$ otherwise. Now we describe the three phases of TPUT.

1. Phase 1: Each node sends its top-k objects to the central manager. The central manager then calculates the partial sums for all objects seen so far and identifies the objects with the k highest partial sums.

2. Phase 2: Let τ_1 be the partial sum of the kth object. This value is referred to as the "phase-1 bottom". The central manager first sends a threshold value $T = \tau_1/m$ to every node in the system. Then each node sends its objects to the central manager, whose scores are no less than T. The intuition is that if an object is not reported by any node, its sum must be less than τ_1. Hence it cannot be a top-k object. Now the central manager can re-calculate the lower bound. It calculates the new partial sums for the objects seen so far. Then the new lower bound τ_2 ("phase-2 bottom") is the partial sum of the kth object. An upper bound of each object's aggregated score is calculated by $U_{\texttt{sum}}(O) = S'_1(O) + \ldots + S'_m(0)$ where $S'_i(O) = S_i(O)$ if O has been reported by node i, and $S'_i(O) = T$ otherwise. If the upper bound of an object's aggregate score is less than τ_2, it can be pruned. After pruning, the set of objects left are the top-k object candidates.

3. Phase 3: This phase identifies the top-k objects. The central manager sends the top-k object candidate set to each node and each node in turn sends the scores of these objects to the central manager. Hence, the central manager can calculate the real scores for these objects and then identify the exact top-k objects.

Example 2 *Consider the lists shown in Figure 1 again and the top-k query still requests the top 2 objects. In the first phase, each node sends objects with scores at positions 1 and 2 to the central manager. The central manager calculates the partial sums:* $S_{\texttt{psum}}(O_5) = 21$, $S_{\texttt{psum}}(O_4) = 48$, $S_{\texttt{psum}}(O_3) = 30$, $S_{\texttt{psum}}(O_2) = 17$, $S_{\texttt{psum}}(O_1) = 29$. *The two highest partial sums are 48 and 30. The phase-1 bottom* τ_1 *is 30. Thus the threshold value* T *used in phase 2 is set to* $30/3 = 10$. *In phase 2, node 1 sends* $\langle object, score \rangle$ *pairs up to position 6, node 2 up to position 4, and node 3 up to position 2. The central manager calculates the partial sums for the objects and calculates the phase 2 bottom,* τ_2, *as* 59 *since the top 2 partial sums are* $S_{\texttt{psum}}(O_3) = 67$ *and* $S_{\texttt{psum}}(O_4) = 59$. *Moreover objects* O_0, O_1, O_2, O_5, O_6 *and* O_7 *are pruned as their upper bounds are less than* τ_2. *Now the top 2 object candidate set* $S_{\texttt{candidate}}$ *is* $\{O_4, O_3\}$. *In phase 3, the central manager sends* $S_{\texttt{candidate}}$ *to each node. Then each node sends the scores of these candidate objects to the central manager. In the end, the central manager identifies the top 2 objects as* O_3 *and* O_4.

4. New Ranking Aggregation Algorithms

In this section, we propose three new algorithms for answering top-k queries in distributed systems. The first algorithm, the *Three-Phase Adaptive-Threshold* algorithm (TPAT), generalizes TPUT by exploiting data distributions using summary statistics to further enhance the pruning power of TPUT. The second algorithm, the *Three-Phase Object-Ranking* based algorithm (TPOR), prunes ineligible objects by their rankings (positions). In contrast, TPUT prunes ineligible objects based on their scores. The last algorithm, the *Hybrid-Threshold* algorithm (HT), combines the advantages of both TPUT and TPOR, and demonstrates that it is very robust to different data distributions.

4.1. Three-Phase Adaptive-Threshold Algorithm

In this section, we extend TPUT by relaxing the condition on how to divide the phase-1 bottom τ_1 among all nodes. By dividing the phase-1 bottom τ_1 uniformly among the nodes, TPUT assumes object scores are uniformly distributed among nodes in the network, i.e., each node contributes approximately the same to the result set of top-k objects. However this assumption does not consider the case in the real world where some nodes in the systems are hot spots for content-sharing. This results in non-uniformly distributed data among nodes. That is, some nodes may have objects with larger score distributions while other nodes may have objects with smaller score distributions. For convenience, they are referred to as *hot nodes* and *cold nodes* respectively. The probability for a top-k object being from a hot node is much higher than being from a cold node. Intuitively, hot nodes usually contribute a larger portion of top-k objects than cold nodes do. Hence, we propose to divide τ_1 to nodes adaptively according to their data distributions. In general a threshold lower than τ_1/m for a hot node allows more objects to be sent to the central manager, and vice versa. This adaptive division of phase-1 bottom τ_1 still guarantees the true top-k objects are among all the objects returned by nodes in phase 2, which is summarized in Theorem 1.

Theorem 1 *Non-uniformly dividing the phase-1 bottom* τ_1 *into thresholds* $T_1, \ldots, T_m$ *assigned to node 1, ..., node* m *respectively such that* $\sum_{i=1}^{i=m} T_i = \tau_1$ *still guarantees that the true top-k objects are among the objects sent from nodes in phase 2.*

Proof: Assume O is an object which is not reported to the central manager by any node. That is, its score in node i, s_i, must be smaller than T_i. Thus, its aggregated score must be smaller than τ_1. Since there are already k objects whose partial sums are no less than τ_1, O cannot be a top-k object. Thus the true top-k objects must be among all the objects sent from nodes in phase 2. □

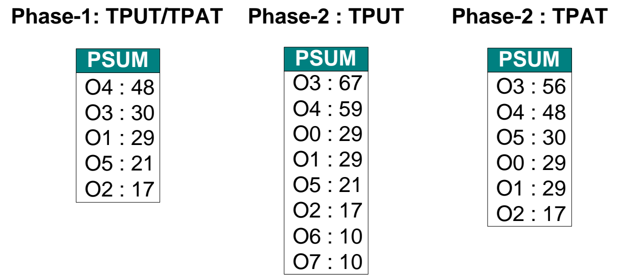

Figure 2. Phase 1 & 2 of both TPUT and TPAT

We illustrate the adaptive division of phase-1 bottom τ_1 by using the example lists shown in Figure 1. Assume that the central manager asks for a top 2 query. From Figure 2, we observe that node 2 has objects with a larger score distribution as compared to node 1 and node 3. Hence node 2 plays an important role to the final scores of top-k objects in the result set with higher probability than the other nodes. Thus more objects are expected to be sent from node 2 to the central manager and fewer objects from node 1 or node 3. If, on the other hand, τ_1 is non-uniformly divided into $T_1 = 12, T_2 = 8$, and $T_3 = 10$, which are assigned to nodes 1, 2, 3 respectively as the thresholds. Then, in phase 2, node 1 sends $\langle object, score \rangle$ pairs up to position 2, node 2 up to position 5, and node 3 up to position 2. As compared with the uniform threshold, non-uniform thresholds send 3 fewer number of $\langle object, score \rangle$ pairs. Furthermore, we can observe that the size of the partial sum lists calculated in phase

2 for the non-uniform threshold case is smaller than that for the uniform threshold case. Therefore, at the end of phase 2, there is a smaller candidate set and this in turn reduces the number of objects sent through the network. Figure 2 shows the number of $\langle object, score \rangle$ pairs sent in the first two phases of both TPUT and TPAT.

TPAT algorithm is summarized as follows:

1. Phase 1: same as TPUT.
2. Phase 2: Instead of using a uniform threshold $T = \tau_1/m$, the central manager divides τ_1 non-uniformly into $T_1, \ldots, T_m$ according to some summary statistics sent from nodes. Then it sends $T_i, \ldots, T_m$ to node i, ..., node m respectively as their thresholds. The rest is the same as TPUT except that the upper bound of each object's aggregated score calculated by $U_{\texttt{sum}}(O) = S'_1(O) + \ldots + S'_m(0)$ where $S'_i(O) = S_i(O)$ if O has been reported by node i, and $S'_i(O) = T_i$ otherwise.
3. Phase 3: same as TPUT.

Theorem 2 *The TPAT algorithm correctly returns the exact top-k objects for any data distribution in each node of a two-tier distributed system.*

Proof: To prove the correctness of TPAT, first we need to prove that the objects returned from each node whose scores are no less than their assigned thresholds guarantee that top-k objects are among them. This is established by Theorem 1. Second, we need to prove that any object pruned by the upper bound calculation in Phase 2 cannot be a top-k object. If the upper bound of an object's aggregated score is less than τ_2, it cannot be a top-k object since there are already k objects whose partial sums are no less than τ_2. Hence we proved the correctness of TPAT. □

The motivating example above only develops the general framework for an adaptive division of the phase-1 bottom τ_1 according to the data distribution in the network to help reduce bandwidth consumption. The main challenge is how the data distribution can be captured approximately using summary statistics and how they are used to guide the adaptive division of τ_1.

Since histograms have been widely used in various database problems and are the most commonly used form of statistics in practice, we now investigate them as a tool to guide the adaptive division of τ_1. The main advantages of histograms over other techniques are that they incur almost no run-time overhead and for real database applications, they provide low-error estimate while requiring minimal space [12]. *Equi-depth histograms* [12] are used to illustrate the framework of our proposed technique. Note that any kind of histograms can fit in our framework. Equi-depth histograms are constructed by dividing the domain into b buckets with roughly the same number of tuples in each bucket. This number and the bucket boundaries are stored. Equi-depth histograms usually handle skewed data well with relatively low estimation error. For notational convenience, the equi-depth histogram for the data in node i is represented by $H_i = \{B^i_1, \ldots, B^i_{b_i}\}$. $B^i_j (1 \le j \le b_i)$ is in the form of $([V^i_{min}, V^i_{max}], f)$, where $[V^i_{min}, V^i_{max}]$ is the boundary of bucket B^i_j and f is the total number of objects in the bucket. Hence, given a score predicate $s \ge p$, the total number of objects within the range can be approximated by examining the overlapped buckets by assuming a uniform distribution in each bucket.

In general, we need to divide the phase-1 bottom τ_1 among the different histograms of the different nodes so as to minimize the number of objects retrieved from each node. This is a linear programming problem, whose complexity increases as the number of buckets in the histogram per node increases. To simplify the problem, we consider one bucket histograms. Now the histogram for node i is represented by $H_i = \{B^i_1 = ([V^i_{min}, V^i_{max}], f^i)\}$, i.e., each node returns the score range of its objects and the total number of objects at the node. Assume τ_1 is divided into $T_1, \ldots, T_m$ such that $\sum_{i=1}^m T_i = \tau_1$. Assuming uniform score distribution at each node, we can approximate the number of objects whose scores are no less than T_i by using the score range and the number of objects, i.e., $\frac{f^i}{V^i_{max} - V^i_{min}} * (V^i_{max} - T_i)$. Assume that this number is $f_i(T_i)$. Hence, the goal of adaptive division of τ_1 into $T_1, \ldots, T_m$ is to minimize

$$\sum_{i=1}^m f_i(T_i) = \sum_{i=1}^m \frac{f^i}{V^i_{max} - V^i_{min}} * (V^i_{max} - T_i)$$

such that $\sum_{i=1}^m T_i = \tau_1$ and $V^i_{min} \le T_i \le V^i_{max}$.

k	5	10	25	50	100
TPAT	37125	37188	37599	38406	40417
TPUT	90894	91341	93111	94446	96080
Improved	53769	54153	55512	56040	55663
Percentage	59.16%	59.29%	59.62%	59.34%	57.93%

Figure 3. Performance comparisons between TPAT and TPUT

We generated a set of synthetic data sets by varying the number of nodes and the number of objects in each node. They are chosen to simulate the hot and cold nodes and the score distribution in each node is uniform. From the experimental results, we observed that TPAT, as compared to TPUT, reduces the number of $\langle object, score \rangle$ pairs transmitted by 10%-60%. Figure 3 shows one of the experimental results we obtained for a data set with 20 nodes and the number of objects for each node ranges from 500-10000.

However, when the score distribution in each node is non-uniform, the selectivity estimation accuracy using only

one bucket for each histogram cannot be guaranteed. Using more than one bucket for each histogram to summarize score distribution makes the optimization much more complex. Since we do not know which bucket T_i will fall in, one optimization function for the linear programming problem is not enough. In the worst case, T_i may lie within the boundary of any bucket and thus, for m nodes, we will have $b_1 * b_2 * \ldots * b_m$ optimization functions. which is computationally infeasible due to the large number of nodes.

The main contribution of TPAT is to generalize TPUT and exploit data distributions among nodes to further enhance the pruning power by non-uniformly dividing the phase-1 bottom. However, when the data distribution in each node is not uniform, to accurately estimate data distributions incurs efficiency concern. Hence, we introduce alternative techniques without using a *priori* knowledge on data distributions.

4.2. Three-Phase Object-Ranking Based Algorithm

We now propose a new algorithm, referred to as *Three-Phase Object-Ranking* based algorithm (TPOR), that is more likely to capture the heterogeneous nature of distributed networks without using any summary statistics. Its pruning of ineligible objects is based on object rankings instead of their scores. In particular, in the second phase of this new algorithm, instead of assigning a threshold for each node, the central manager sends the current top-k *object list* to each node. Upon receiving this "top-k" object list, each node examines its objects and passes all of its local objects that are ranked higher than any of the objects in the list to the central manager. In this way, the correlation between the object score and the object ranking are captured, which can avoid the case where an inappropriately small phase-1 bottom τ_1 is obtained by TPUT. In the following we use an example to show how TPOR works.

Example 3 *We again consider the lists shown in Figure 1 and the top-k query still requests the top 2 objects. As described in Example 2, after the first phase, the objects with the two highest partial sums are O_4 and O_3, which can be also seen from Figure 2. Hence in phase 2, the central manager sends the set of objects $\{O_4, O_3\}$ to each node. The lowest ranking of these two objects in node 1, 2, 3 are 4, 4 and 2 respectively. Therefore, node 1 sends $\{O_5, O_2, O_4, O_3\}$, node 2 sends $\{O_4, O_1, O_0, O_3\}$, and node 3 sends $\{O_3, O_4\}$. As compared with TPUT, 2 fewer objects are needed to be sent to the central manager. The central manager calculates the partial sums for the objects and identifies the phase 2 bottom, τ_2, as 59 since the top 2 partial sums are $S_{\texttt{psum}}(O_3) = 67$ and $S_{\texttt{psum}}(O_4) = 59$. Furthermore objects O_0, O_1, and O_2 are pruned as $U_{\texttt{sum}}(O_0) = 54$, $U_{\texttt{sum}}(O_1) = 54$, and $U_{\texttt{sum}}(O_2) = 57$ are less than τ_2. Now the top 2 object candidate set $S_{\texttt{candidate}}$ is $\{O_3, O_4, O_5\}$. In phase 3, the central manager sends $S_{\texttt{candidate}}$ to each node. Then each node sends the scores of these candidate objects to the central manager. In the end, the central manager identifies the top 2 objects as O_3 and O_4.*

The *Three-Phase Object-Ranking* based algorithm (TPOR) is summarized as follows.

1. Phase 1: same as TPUT.
2. Phase 2: The central manager broadcasts the list L of the top-k object IDs from the partial sum list to all the nodes in the network.

   Upon receiving the list L, for each object O_j in L, node i finds its local score $V_{i,j}$ (if O_j does not occur in the local list, $V_{i,j} = 0$) and determines the lowest local score T_i among all the k objects in L. Then node i sends the list of local objects whose values are $\geq T_i$ to the central manager.

   Now the central manager calculates the partial sums of all the objects seen so far, and identifies the objects with the k highest partial sums. Let us call the kth highest partial sum "phase-2 bottom" and denote it by τ_2. Then the central manager tries to prune away more objects. It calculates the upper bounds of the objects seen so far using $U_{\texttt{sum}}(O) = S'_1(O) + \ldots + S'_m(0)$ where $S'_i(O) = S_i(O)$ if O has been reported by node i, and $S'_i(O) = T_i$ otherwise. Then the central manager removes any object O_j from the candidate set whose upper bound is less than τ_2.
3. Phase 3: same as TPUT.

Theorem 3 *The TPOR algorithm correctly returns the exact top-k objects for any data distribution in each node of a two-tier distributed system.*

Proof: To prove the correctness of TPOR, first we need to prove that the objects returned from each node in phase 2 guarantee that the top-k objects are among them. This can be established as follows. If an object O is not reported by a node, there are at least k objects from list L whose scores are greater than its score in any of the node. Hence, O cannot be a top-k object. Second, we need to prove that any object pruned by the upper bound calculation in phase 2 cannot be a top-k object. This is similar to that of Theorem 2. □

The difference between TPOR and TPUT lies in the fact that in TPOR, during phase 1, the central manager sends the entire top-k object ID list to all the nodes. We argue this will not incur much overhead since, in practice, the object ID can be hashed to integers, the value of k is in general

not large and the object ID list can be multicast to all the nodes simultaneously. However, similar to TPUT, TPOR also depends on the data distribution. For example, in phase 2, if one node does not have any object in the object ID list calculated in phase 1, it will send all its local objects to the central manager.

4.3. Hybrid-Threshold Algorithm

Since distributed systems have the heterogeneous nature and the number of nodes in them is usually large, the uniform threshold calculated by TPUT for each node to prune objects may be very small. This results in many more objects returned from all the nodes. Alternatively, if an object in the object ID list calculated by TPOR in phase 1 ranks very low in some nodes or even does not appear, then those nodes will send almost their entire list to the central manager. In this subsection, we propose a hybrid algorithm, the *Hybrid-Threshold* algorithm (HT), which tries to combine the advantages of both TPOR and TPUT. In the second phase of HT, the central manager asks each node to send objects whose scores are no less than a hybrid threshold, which is calculated as the maximum of the uniform threshold $T = \tau_1/m$ from TPUT and the threshold obtained by TPOR. However, this cannot guarantee the correctness of the algorithm. It is possible that some objects in a node whose scores are between the uniform threshold by TPUT and the threshold by TPOR, are top-k objects. Thus, we devise to add a patch phase in order to make the algorithm correctly return the top-k objects. After phase 2, the central manager calculates the new partial sums for all the objects seen so far and identifies the objects with the k highest partial sums. Let the kth partial sum denote τ_2. Then the central manager calculates $T_{patch} = \tau_2/m$. Since after phase 2, the central manager knows the lower bounds of the object scores of nodes, denoted as $T_1, \ldots, T_m$. If $T_{patch} \leq T_i$, the central manager sends T_{patch} to node i and asks node i to send the objects whose scores are no less than T_{patch}. Since T_{patch} is greater than T, calculated in the beginning of phase 2, the total number of objects sent by HT is no greater than that of TPUT. However, if $T_{patch} > T_i$ for every i, there is no need for this patch phase, i.e., all top-k object candidates have been considered. The *Hybrid-Threshold* algorithm is summarized as follows:

1. Phase 1: same as TPUT.

2. Phase 2: The central manager broadcasts the list L to all the nodes in the network and $T = \tau_1/m$ as well.

   Upon receiving the list L, for each object O_j in L, node i finds its local score $V_{i,j}$ (if O_j does not occur in the local list, $V_{i,j} = 0$) and determines the lowest local score S^i_{lowest} among all the k objects in L. Then node i sends the list of local objects whose values are $\geq T_i = \max(S^i_{lowest}, T)$ to the central manager.

   Now the central manager calculates the partial sums for all the objects seen so far, and identifies the objects with the k highest partial sums. Let us call the kth highest partial sum "phase-2 bottom" and denote it by τ_2.

3. Phase 3 (patch phase if necessary) : The central manager checks if the threshold from node i, T_i in phase 2 is greater than $T_{patch} = \tau_2/m$. If so, the central manager will send T_{patch} to node i as the threshold and ask it to send all the objects whose scores are no less than T_{patch}.

   Now the central manager calculates the partial sums for all the objects seen so far, and identifies the objects with the k highest partial sums. Let us call the kth highest partial sum "phase-3 bottom" and denote it by τ_3.

   Then the central manager tries to prune away more objects. It calculates the upper bounds of the objects seen so far using $U_{\texttt{sum}}(O) = S'_1(O) + \ldots + S'_m(0)$ where $S'_i(O) = S_i(O)$ if O has been reported by node i, and $S'_i(O) = \min(T_i, T_{patch})$ otherwise. Then the central manager removes any object O_j from the candidate set whose upper bound is less than τ_3.

4. Phase 4: same as TPUT.

Theorem 4 *The HT algorithm correctly returns the exact top-k objects for any data distribution in each node of a two-tier distributed system.*

Proof: The proof is similar to that for the correctness of TPAT. □

5. Experimental Evaluation

In this section, we experimentally evaluate the performance of our proposed algorithms TPOR and HT. Note that TPAT is not included here due to the computational overhead of using multi-bin histograms. However, TPAT is significant in that it provides us the basic framework which enables us to develop TPOR and HT. We implemented TPUT, TPOR, and HT in Java and compared their performance over various synthetic and real data sets. Since TPUT outperforms TA in most cases as shown in [4], we do not compare our proposed algorithms with TA. The performance metric we use for the algorithms is the bandwidth consumption. We are mainly concerned with the number of $\langle object, score\rangle$ pairs sent from nodes to the central manager since it is the dominant factor in bandwidth consumption. The control messages from the central manager to the

nodes are broadcast through a broadcast media. Their size is very small and hence can be ignored.

5.1. Synthetic Data Sets

We generated various synthetic data sets for performance evaluation of our proposed algorithms. These synthetic data sets are generated as follows. Assume there are m nodes, node 0, ..., $m-1$, in the network and each node has n objects. Initially n values $v_1, \ldots, v_n$ are generated, which follow the Zipf's distribution [16] with a Zipf factor α. These n values are assigned to the n objects as their scores in node 0. The scores of an object O in other nodes are generated by using a random walk model:

$$S[i] = S[i-1] + s_i$$

$S[i]$ represents the score of object O at node i. s_i is a random number in the range $[-r, +r]$. r is set to $c \times S[0]$ where c is a constant which is less than or equal to 10%. By varying α and c, we can simulate different scenarios such as the scenario in which the object rankings are similar in different nodes or the scenario in which the object rankings vary in different nodes.

The experimental results in this section are based on five synthetic data sets. Each of them has $m = 100$ nodes and each node has 10000 objects. These five data sets have $\alpha = 0.1, 0.3, 0.5, 0.8, 1.0$ respectively. They are referred to as *Synthetic-α*. Synthetic-0.1 simulates a scenario where the rankings of objects in each node are quite different. Since $\alpha = 0.1$, the initial scores generated for objects are less skewed. Also the constant c of the random walk model is set to a larger value for those objects which have lower scores in node 0 and a smaller value for those objects which have higher scores in node 0. This ensures that some objects with higher initial rankings have lower rankings in other nodes and vice versa. With α increasing, the rankings of objects among all nodes tend to be similar. When $\alpha = 1$, the initial scores generated for objects are quite skewed. Moreover, since c is at most 10%, it is highly probable that the initial rankings of objects remain approximately the same for the other nodes. Thus, Synthetic-1.0 simulates a scenario in which object rankings are very similar in different nodes.

In the following, we present the performance comparisons of TPUT, TPOR, and HT over synthetic data sets. The queries are for the top-k referenced objects. We ran extensive experiments over each data set by varying k from 5 to 100 and only report the results for $k = 5, 10, 25, 50, 100$. Figures 4 and 5 show the performance comparisons over Synthetic-0.1 and Synthetic-0.5 data sets respectively. We have the following observations.

- TPOR and HT outperform TPUT, and the improvement of TPOR and HT is significant. On average TPUT sends 2 to 3 times more number of $\langle object, score \rangle$ pairs than TPOR and HT do.
- When the object rankings among nodes become more similar, i.e., Figure 5 where $\alpha = 0.5$, the performance of TPOR and HT over them becomes relatively stable when k increases while this is not the case with TPUT. This is because, for such cases, TPOR and HT prune objects mainly based on their rankings, which is less sensitive to the score variations of objects.
- For data sets in which the object rankings among nodes are less similar, i.e., Figure 4 where $\alpha = 0.1$, when k increases, the improvement of HT over TPOR decreases. This is because a higher k results in more objects sent to the central manager. Thus, the object ID list calculated for TPOR more accurately captures the true top-k objects. Then more nodes in HT use the thresholds calculated by the object rankings instead of the uniform threshold calculated by TPUT. Therefore, fewer $\langle object, score \rangle$ pairs are eliminated by using the uniform threshold.
- The less the object rankings among nodes are similar, the more $\langle object, score \rangle$ pairs are eliminated by HT as compared with TPOR. The reason is that for data sets in which the object rankings are less similar, TPOR may calculate a less accurate object ID list and send more objects in the second phase. However, HT combines the advantages of both TPUT and TPOR, which can lead to significant gains.

Figure 6 examines the effect of the Zipf factor on the performance of our algorithms where k is set to 50. As α increases, the object rankings in different nodes become more similar and the number of $\langle object, score \rangle$ pairs sent by TPUT, TPOR and HT decreases. This is because the objects collected from the first phase provide more accurate information for pruning. Also, the improvement of TPOR and HT over TPUT becomes more pronounced. In Figure 6, for Synthetic-0.8 and Synthetic-1.0 data sets, TPOR outperforms HT. This is because these two datasets have very similar object rankings in different nodes and hence, most thresholds calculated in the second phase are actually greater than the threshold calculated by the TPUT method. Nevertheless, HT requires the patch phase which may have a lower threshold and thus more objects are sent during the patch phase.

5.2. Real Data Set

We studied the performance of the algorithms on a real data set containing the 2 hour URL access log from the 29 servers hosting the website for the 1998 World Cup Soccer

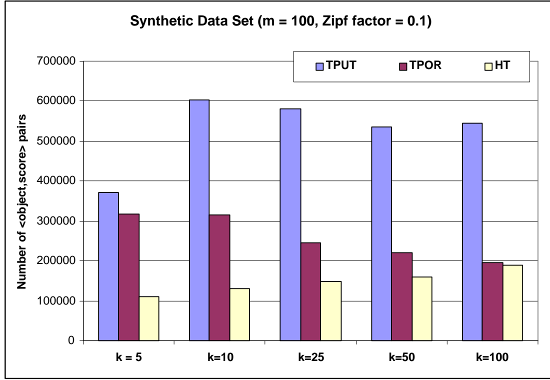

Figure 4. Performance comparisons over the Synthetic-0.1 data set

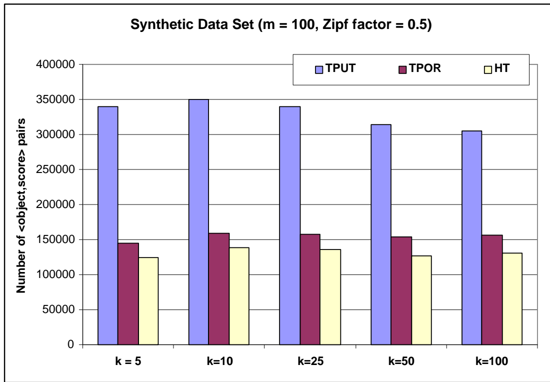

Figure 5. Performance comparisons over the Synthetic-0.5 data set

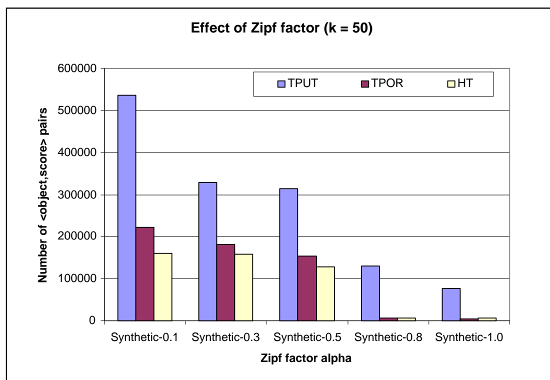

Figure 6. Examine effect of Zipf factor α

on June 18, 1998. It is referred to as *WorldCup98-29*. This data set contains 29 nodes and the zipped size is 759KB.

In the following, we discuss the performance comparisons of TPUT, TPOR, and HT over the real data set in Figure 7. The queries are for the top-k referenced URLs. Also note that we ran extensive experiments over the data set by varying k from 5 to 100 and only report the results for $k = 5, 10, 25, 50, 100$.

Figure 7 shows the performance comparisons over the WorldCup98-29 data set. TPOR and HT outperform TPUT in most cases. The saving in bandwidth consumption for $k = 10$ is significant and up to 75%. The reason is that, for the WorldCup98-29 case, the final top 10 objects have very high rankings in all nodes. Thus, TPOR and HT can easily avoid returning ineligible objects, which are possibly returned by TPUT because of the lower value of τ_1. TPOR and HT perform approximately the same in most cases except for the top 5 case. This is because, in the first phase of TPOR, each node in the distributed system only returns its local top 5 objects to the central manager. The number of objects returned from all nodes is not sufficient to capture the final top-k objects. Alternatively, some objects which actually rank very low in some nodes are included in the object ID list which is calculated in the first phase. This in turn results in some nodes returning too many objects.

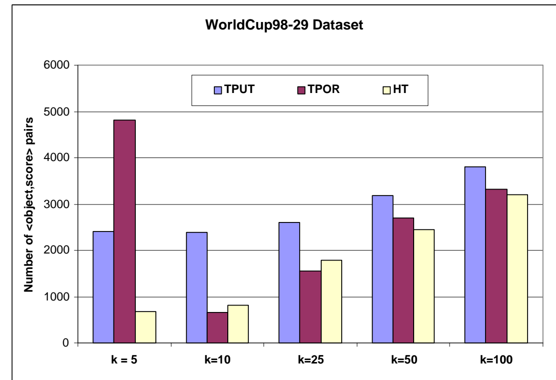

Figure 7. Performance comparisons over WorldCup98-29 data set

6. Conclusion and Future Work

In this paper, top-k query calculation in distributed networks is studied. Prior research on distributed top-k query calculation did not take into account data distributions when pruning ineligible objects. Non-uniformity of data distributions is likely to occur frequently due to the heterogeneous nature of distributed systems. In this paper, we proposed three different distributed top-k query algorithms that consider data distributions in different ways. We performed extensive experiments over both real and synthetic data sets to evaluate our proposed algorithms as compared with prior

research. Our experimental results demonstrate that our final algorithm, HT, is more suitable for answering top-k queries in distributed systems when dealing with data with different distributions. So far, we only considered two-tier distributed systems. One natural step for our future work is to study the top-k query problem over distributed systems with hierarchical structures such as peer-to-peer systems.

References

[1] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. of Intl. Conf. on Managment of Data (SIGMOD)*, pages 563–574, 2003.

[2] W-T. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive distributed top-k retrieval in peer-to-peer networks. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, 2005, to appear.

[3] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. on Database Systems*, 27(2):153–187, 2002.

[4] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proc. of Intl. Symposium on Principles Of Distributed Computing (PODC)*, pages 206–215, 2004.

[5] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, pages 397–410, 1999.

[6] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. of Intl. Symp. on Principles of Database Systems (PODS)*, pages 216–226, 1996.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of Intl. Symposium on Principles of Database Systems (PODS)*, pages 102–113, 2001.

[8] U. Guntzer, W-T. Balke, and W. Kiessling. Optimizing multi-feature queries in image databases. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, pages 419–428, 2000.

[9] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *Proc. of Intl. Conf. on Very Large Data Base (VLDB)*, pages 754–765, 2003.

[10] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmi. Rank-aware query optimization. In *Proc. of Intl. Conf. on Managment of Data (SIGMOD)*, pages 203–214, 2004.

[11] S. Nepal and M.V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, pages 22–31, 1999.

[12] W. Poosala, P.J. Haas, Y.E. Ioannidis, and E.J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. of Intl. Conf. on Management of Data (SIGMOD)*, pages 294–305, 1996.

[13] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, pages 648–659, 2004.

[14] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, pages 277–288, 2003.

[15] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, pages 189–200, 2003.

[16] G. K. Zipf. *Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology*. Addison-Wesley, 1949.