# Efficient Top-K Query Calculation in Distributed Networks

Pei Cao
cao@theory.stanford.edu

Zhe Wang
Department of Computer Science
Princeton University
Princeton, NJ 08540
zhewang@cs.princeton.edu

## Abstract

This paper presents a new algorithm to answer top-$k$ queries (e.g. "find the $k$ objects with the highest aggregate values") in a distributed network. Existing algorithms such as the Threshold Algorithm [FLN01] consume an excessive amount of bandwidth when the number of nodes, $m$, is high. We propose a new algorithm called "Three-Phase Uniform Threshold" (TPUT). TPUT reduces network bandwidth consumption by pruning away ineligible objects, and terminates in three round-trips regardless of data input.

The paper presents two sets of results about TPUT. First, trace-driven simulations show that, depending on the size of the network, TPUT reduces network traffic by one to two orders of magnitude compared to existing algorithms. Second, TPUT is proven to be instance-optimal on data series that satisfy a lower bound on the slope of decreases in values. In particular, analysis shows that by using a pruning parameter $\alpha < 1$, TPUT achieves a qualitative reduction in network traffic, for example, lowering the optimality ratio from $O(m * m)$ to $O(m * \sqrt{m})$ for data series following Zipf distribution.

## 1 Introduction

We investigate algorithms that answer "top-$k$" queries efficiently in distributed networks. The performance criteria are low latency and low bandwidth consumption. Such algorithms are important in a variety of systems; our particular interest is content distribution networks for large enterprises.

Large enterprises have branch offices located around the globe. The offices are usually connected to the enterprise data center via WAN links in a star topology. The number of branch offices ranges from a few tens to a few thousands. Due to the diverse geographical locations of branch offices, the links between the offices and the data center have low bandwidth, typically 128Kb/s to 2Mb/s.

To enable Web and streaming media applications at the branch offices, a Content Distribution Network (CDN) is usually deployed. In the CDN, a "content engine" is installed at each branch office. The device acts as a Web cache, a streaming media cache, and a server that serves pre-positioned Web and video contents. The devices are managed by a central management station located at the data center.

Successful operations of CDN rely on effective monitoring of the activities on the network, which means that the central management station is often asked to answer "top-$k$" queries. For example, the administrator routinely asks for "list the top-$k$ most popular URLs across the whole CDN", or "list the objects whose total byte savings across all caches are among the top $k$." Naive methods for answering these queries would have each cache send data about all objects to the central manager. Since the number of objects at each cache easily runs to millions, the sheer amount of data can consume excessive WAN bandwidth and defeat the bandwidth-saving purpose of the CDN. Hence, more sophisticated algorithms are needed.

In this paper, we present a new algorithm, *Three-Phase Uniform Threshold (TPUT)*, that answers the "top-$k$" queries in large-scale networks efficiently.

### 1.1 Problem Definition

Assume that there are $m$ data series, each data series $d$ is a list of $\langle x, v_d(x)\rangle$ pairs, where $x$ is an object and $v_d(x) > 0$ is the value of the object. The list is sorted by object values, from the highest to the lowest; thus we also call a data series a "sorted list". The sets of objects in each data series overlap but are not identical. If an object $y$ doesn't appear in data series $d$, we say that $v_d(y) = 0$.

For each object $x$, one can calculate its aggregate value across the $m$ data series: $V(x) = v_1(x) +$

$v_2(x) + \ldots + v_m(x)$. The query is to find $k$ objects, $x_1, x_2, ..., x_k$, whose values $V(x_1), V(x_2), ..., V(x_k)$ are the highest $k$ values among all objects.

In a network of $m$ nodes connected to a central manager, each node is a data series. The goal of the algorithm is to answer the query with minimum amount of communication between the nodes and the central manager.

As described, the problem is a generic one that can be found in almost any monitoring network or collaborative distributed system. When the lists are short, the simple method where each node sends its list to the manager works fine. However, when the lists are long, more sophisticated solutions are needed. Such solutions are useful in many systems besides content distribution networks, for example, sensor networks and spam detection networks.

## 1.2 Review of the Threshold Algorithm

Database communities have studied various methods to evaluate top-$k$ queries [NR99, Fag99, UBK00, BGM02, FLN01, BO03]. The queries in the studies aggregate values over a few databases. The arguably best algorithm is the Threshold Algorithm (TA), discovered independently by multiple groups [NR99, UBK00, FLN01] and examined thoroughly in [FLN01].

Briefly, TA goes down the sorted lists in parallel, one position at a time, and calculates the sum of the values at that position across all the lists. This sum is called the "threshold" in [FLN01] and the "stopping value" in this paper. Everytime a new object appears, TA looks up in all lists to find its aggregate value. TA stops when it finds $k$ objects whose values are higher than the stopping value. The algorithm is correct because any object that it has not seen cannot have a value higher than the stopping value.

As an example, look at the data series in Table 1. Assume the query is to find the top 2 objects, i.e. $k = 2$. TA first looks at the objects in position 1 of all lists, which are $O_1$, $O_2$, and $O_3$. It looks them up in all lists and finds their aggregate values, $V(O_1) = v_1(O_1)+v_2(O_1)+v_3(O_1) = 10+1+9 = 20$, $V(O_2) = v_1(O_2) + v_2(O_2) + v_3(O_2) = 7 + 10 + 1 = 18$, and $V(O_3) = v_1(O_3)+v_2(O_3)+v_3(O_3) = 8+2+10 = 20$. The stopping value at position 1 is $10+10+10 = 30$. Hence the algorithm cannot stop and must go onto position 2. At position 2 the objects are $O_3$, $O_4$, and $O_1$. $O_4$ is a new object so the algorithm finds its aggregate value $V(O_4) = 5 + 9 + 5 = 19$. The stopping value at position 2 is $8+9+9 = 26$ and the algorithm must go on. The algorithm finally stops at position 5 and concludes that the top 2 objects are $O_6$ with value 22 and $O_1$ with value 20.

| Position | Series 1 | Series 2 | Series 3 |
|---|---|---|---|
| 1 | $\langle O_1, 10\rangle$ | $\langle O_2, 10\rangle$ | $\langle O_3, 10\rangle$ |
| 2 | $\langle O_3, 8\rangle$ | $\langle O_4, 9\rangle$ | $\langle O_1, 9\rangle$ |
| 3 | $\langle O_5, 8\rangle$ | $\langle O_6, 8\rangle$ | $\langle O_7, 8\rangle$ |
| 4 | $\langle O_6, 8\rangle$ | $\langle O_8, 6\rangle$ | $\langle O_9, 7\rangle$ |
| 5 | $\langle O_2, 7\rangle$ | $\langle O_7, 5\rangle$ | $\langle O_6, 6\rangle$ |
| 6 | $\langle O_4, 5\rangle$ | $\langle O_3, 2\rangle$ | $\langle O_4, 5\rangle$ |
| 7 | $\langle O_9, 1\rangle$ | $\langle O_1, 1\rangle$ | $\langle O_2, 1\rangle$ |

Table 1: An example data set with three data series.

Adapted to running over a network, TA would go through rounds, each round involving two round-trip communications. In the first round-trip, the manager asks nodes about data at a particular position in their lists. In the second round-trip, the manager sends all nodes a list of object IDs and the nodes respond with values of those objects in their lists. To reduce the number of rounds, nodes can send data from a block of positions each time. The size of the block affects the number of rounds before TA terminates, but does not significantly affect the total amount of network traffic. In this paper we assume the size of the block is $k$, that is, each round goes $k$ positions down the lists.

TA works well when the number of nodes, $m$, is small. However, when $m$ is large, the network traffic involved in the second round-trip can become excessive. Unless each object's positions in the sorted lists are very similar (i.e. if an object appears in position $k$ in one list, then it appears in positions near $k$ in other lists), the number of unique objects reported in the first round-trip is $O(m)$. These objects are then looked up in all $m$ nodes, leading to overhead of $O(m^2)$. Indeed, when $m$ is large, the traffic incurred by the central manager to send the list of objects to all nodes can be high enough that the naive algorithm would consume less bandwidth than TA. While one can argue that the factor of $O(m^2)$ is unavoidable in any algorithm's worst-case performances (proved in [FLN01]), TA's worst-cases happen too often in practice.

An additional problem is that the latency of TA is unpredictable because the number of rounds varies by data input. For distributed networks, it's desirable to have an algorithm that terminates in a fixed number of round trips.

### 1.3 Our Algorithm: TPUT

We design the TPUT algorithm to terminate in a fixed number of round trips regardless of input and always give accurate answers. The algorithm executes in three steps:

1. determine a lower-bound estimate for the $k$'th value;

2. use the estimate to prune away ineligible objects as much as possible;

3. look up the resulting set of objects in all nodes to identify the top-$k$ objects.

It draws its efficiency over the naive algorithm and TA through its effective pruning mechanism.

We demonstrate the practical performance of TPUT through trace-driven simulations. Using Web access trace data, we show that the network bandwidth consumption of TPUT can be *one to two orders of magnitude* less than TA, depending on the number of nodes.

We analyze the properties of TPUT following the concept of "instance-optimality" as proposed in [FLN01]. We consider the class of fixed round trip algorithms, of which TPUT is a member. We show that no fixed round-trip algorithm can be instance-optimal when considering all possible data series. However, if data series are limited to data distributions that have a reasonable "slope", then TPUT is instance optimal. Furthermore, by introducing a parameter $\alpha < 1$ in the TPUT algorithm, the performance of the algorithm is improved qualitatively. In the case of Zipf distribution, the optimality ratio is reduced from $O(m * m)$ to $O(m * \sqrt{m})$, where $m$ is the number of nodes in the network.

Finally, we discuss extensions of TPUT to hierarchical networks and peer-to-peer networks.

## 2 Three-Phase Uniform-Threshold Algorithm

Before we describe the TPUT algorithm, we define a few basic operations and notations.

### 2.1 Partial Sums and Upper Bounds

At any stage in the algorithm, the central manager can calculate a *partial sum* of an object $o$, $P(o) = v'_1(o) + v'_2(o) + \ldots + v'_m(o)$, where $v'_i(o) = v_i(o)$ if $o$ has been reported by node $i$, and $v'_i(o) = 0$ otherwise. Since all values are $\geq 0$, an object's partial sum is always a lower bound of its aggregate value, $P(o) \leq V(o)$.

When the central manager receives all objects with values above a certain threshold $T$ from all nodes, it can also calculate an *upper bound* for an object $o$, $U(o) = u'_1(o) + u'_2(o) + \ldots + u'_m(o)$, where $u'_i(o) = v_i(o)$ if $o$ has been reported by node $i$, and $u'_i(o) = T$ otherwise. Clearly, $U(o) \geq V(o)$ for any object $o$.

Assume that the final answer to the top-$k$ query are objects $O_1, O_2, \ldots, O_k$, where $V(O_1) \geq V(O_2) \geq \ldots \geq V(O_k)$. We call this set the "true" top-$k$ objects, and the value $V(O_k)$ the "true bottom", denoted by $\tau$.

### 2.2 Basic Algorithm

TPUT consists of three phases, each taking one round-trip to finish:

- Phase 1: establish a lower bound on the true bottom. The central manager informs all nodes that it would like to initiate calculations of a top-k query. Each node $d$ sends the top $k$ items from its lists.

  After receiving the data from all nodes, the central manager calculates the partial sums of the objects. It then looks at the $k$ highest partial sums, and takes the $k$'th one as the lower bound. We denote this lower bound as $\tau_1$, and call it "phase-1 bottom".

- Phase 2: prune away ineligible objects. The manager now sets a threshold $T = (\tau_1/m)$, and sends it to all nodes. Each node then sends the list of objects whose values are $\geq T$ to the central manager.

  At the end of this round-trip the manager has seen objects in the true top-$k$ set. In other words, if an object is not reported by any node, then its value are $< T$ in all nodes, which means that its aggregate value is $< \tau_1$, and hence it can't be in the top-$k$ set.

  The manager now performs two tasks. First, it refines the lower bound estimate. It calculates a new set of partial sums for the objects, and finds the $k$ highest partial sums. Let's call the $k$'th highest sum "phase-2 bottom", and denote it by $\tau_2$. Clearly, $\tau_1 \leq \tau_2 \leq \tau$.

  Then, it tries to prune away more objects. It calculates upper bounds of the objects as described in the previous section. Objects whose upper bounds are less than $\tau_2$ are eliminated. The set of the remaining objects is the candidate set $S$.

- Phase 3: identify the top-$k$ objects. Now, the manager sends the set $S$ to all nodes, and each node sends the manager the values of objects in $S$. The manager can then calculate the exact sum of objects in $S$, and select the top-$k$ objects from the set. Those objects are the true top-$k$ objects.

As an example, consider the lists in Table 1. In phase 1, all nodes send the data at positions 1 and 2 to the central manager. The manager calculates the partial sums: $V'(O_1) = 19$, $V'(O_2) = 10$, $V'(O_3) = 18$, and $V'(O_4) = 9$. The two highest partial sums are 19 and 18, and the phase-1 bottom $\tau_1$ is 18. Hence, the threshold $T$ is set to $18/3 = 6$. In phase 2, node 1 sends data up to position 5 in its list, node 2 sends data up to position 4, and node 3 sends data up to position 5. The central manager now finds the phase-2 bottom $\tau_2 = 19$ because $V'(O_6) = 22$ and $V'(O_1) = 19$ are now the top 2 sums. Furthermore, $U(O_8) = 18$ and $U(O_9) = 19$, hence, $O_8$ and $O_9$ are eliminated from consideration and $S = O_1, O_2, \ldots, O_7$. In phase 3, the central manager now sends $S$ to all nodes; the nodes respond and the central manager concludes that the top 2 objects are $O_6$ and $O_1$.

**Theorem 2.1.** *The above algorithm correctly identifies the exact top-k objects for any data input.*

*Proof.* As discussed above. □

The above discussion omits two details. First, if a node has sent a piece of data in previous round-trips then it doesn't send it again; the central manager saves all history. Second, at each phase, TPUT always examines the available information and checks if it can safely terminate. For example, in phase 2 if there are $k$ objects whose partial sums are true sums and are higher than the upper bounds of all other objects, TPUT can terminate.

In practice, to guard against the pathological case where a node has a very long sequence of objects with values above the threshold, the central manager can put a limit on the maximum number of objects that a node sends in phase 2. If a node cannot send all objects above the threshold, it should inform the manager. The manager needs to make two adjustments. First, it adjusts the values used in the upper bound calculations. Second, it calculates the sum of the bottom values on the lists sent by the nodes; let's call it $\tau'$. If, at the end of phase 3, the manager finds $k$ objects whose values are $> \tau'$, then the algorithm can terminate. Otherwise, it needs to rerun phase 2 requesting nodes to send all objects above the threshold. Since this mechanism is only needed for pathological cases which are rare in practice, we do not discuss this scheme further.

Finally, note that the algorithm is not limited to sum, and can apply to any strict monotonic aggregation function $f$ [FLN01] as long as there is a way to determine the threshold value $T$ based on the phase-1 bottom. For example, if the function is multiplication, then $T$ would be ${\tau_1}^{1/m}$.

## 2.3 Enhancing the Pruning Power

We can lower the threshold $T$ by setting it to be $(\tau_1/m) * \alpha$, where $0 < \alpha < 1$. We call $\alpha$ the pruning parameter. We choose $\alpha = 0.5$ in our design.

Intuitively, with $T = \tau_1/m * 0.5$, many objects that are reported by a few nodes but whose actual sums are smaller than $\tau_1$ can be detected and eliminated. It turns out that any value of $\alpha < 1$ leads to a qualitative reduction in the size of candidate set $S$. We analyze the choice and impact of $\alpha$ in later sections.

## 2.4 Compression via Hash Arrays

If names of objects are long (e.g. URLs), one can use hash arrays to reduce the amount of traffic. Specifically, in phase 2, when a node sends all objects above $T$ to the manager, it sends in a hash array of counters instead of a list of $\langle url, value\rangle$ pairs. Each entry in the hash array is either 0 if no object hashes into the entry, or the value $V$, where $V$ is the *maximum* of values of objects hashed into the entry. The hash function used and the size of the hash array are the same across all nodes.

Partial sums and upper bounds are calculated on array entries. The candidate set $S$ also consists of array entries. In phase 3, the manager represents $S$ as a bit array and sends it to all nodes, and all nodes respond with lists of objects that hash into entries whose bits are 1.

Due to hash collision, $\tau_2$ might not be a lower bound of $\tau$ anymore. In this case, TPUT will not be able to find $k$ objects whose sum are $\geq \tau_2$. If this happens, the central manager recalculates the candidate set by using $\tau_1$ as the lower bound and the algorithm will then terminate.

To determine the size of the hash array, each node $j$ sends the total number of objects in its sorted list, $n_j$, in phase 1. The central manager then sets the size of the array to be $\Sigma(n_j)$. This creates many empty entries in individual nodes' data, which are easily eliminated by compression. In essence, the hash arrays make sure that each object's name consumes at most three or four bytes in transmission.

**Theorem 2.2.** *The hash array compression does not affect the correctness of the algorithm.*

*Proof.* As discussed above. □

# 3 Experimental Performance of TPUT

We implemented TA, TPUT and TPUT with hash array compression to compare their performance across a range of data sets.

The performance metrics for the algorithms are bandwidth consumption and the number of round trips. For bandwidth consumption we calculate two kinds of byte count:

- "uni-cast" bytes, which assumes that the central manager communicates with each node via unicast. In this case, the central manager constructs individual messages for each node to avoid requesting duplicate information from a node.

- "broadcast" bytes, which assumes that there is a broadcast media, e.g. satellite communication, that the central manager can use to communicate with all nodes. In this case, the central manager constructs one message to broadcast to all nodes, and the message is counted only once.

All information exchanges are compressed via gzip, and the compressed sizes are used in the calculations.

## 3.1 Trace Data

Using web access logs, we simulate the scenario where a central manager talks to a network of web proxies or web servers to execute top-$k$ queries.

- *NLANR-10*: a full-day (Oct 21, 2003) trace log of the 10 caching proxies operated by NLANR [IRC03], which are used by international research communities for accesses to US contents.

- *NLANR-203*: to simulate a higher number of proxies, we split each NLANR proxy's trace into 32 sub-traces, based on the hash of the class A byte of the client IP address. Sub-traces that have fewer than 1000 requested URLs are ignored since they indicate that the corresponding client population are not using the NLANR proxy hierarchy regularly. We are then left with data for a network of 203 proxies.

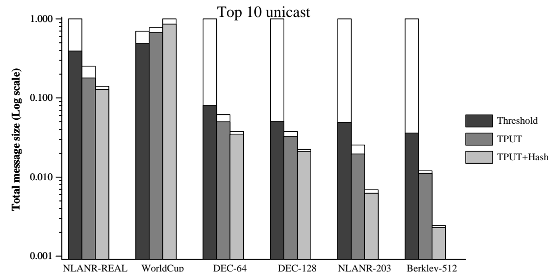


Figure 1: Uni-cast bytes of the algorithms for finding the top 10 objects. *Note that the y-axis is in log scale.*

- *WorldCup-30*: a 2-hour access log from the 30 web servers hosting the web site for the 1998 World Cup Soccer on June 30, 1998.

- *DEC-64*: we take a one-day (Sep 16, 1996) trace from the Digital Equipment Corporation's Internet gateway [Cor96] and split it 64-ways based on client IP addresses. The resulting data simulate the scenarios where the corporate employees are spread among 64 branch offices, with each office having its own gateway proxy.

- *DEC-128*: similar to the DEC-64 data set, we take two days (Sep 16 and 17) worth of traces and split it 128 ways, to simulate the scenario where the number of branch offices is 128.

- *UCB-512*: to understand how the algorithms perform over larger numbers of nodes, we use the 18-day home IP traces gathered by Univ. of California at Berkeley from Nov. 1, 1996 to Nov. 19, 1996. The traces capture 8399 clients' activities, and we split the clients into 512 groups. This simulates the scenario where a large corporation have many small branch offices, each with only ten to twenty people.

The queries are either for top $k$ most referenced URLs, or for top $k$ URLs whose responses have highest aggregate byte count (i.e. consume most bandwidth). We choose two types of queries to vary the value distributions in the input data. Values distributions in the first type of queries tend to be Zipf-like [BCF$^+$99], while value distributions in the second type of queries resemble stair-case functions.

Table 2 summarizes the trace characteristics and lists the actual performance numbers for the naive algorithm and TA.

| Trace name | $m$ | Gzip'ed Size of Data Series | Performance of TA (k=10) unicast | broadcast | # of RTs | Performance of TA (k=100) unicast | broadcast | # of RTs |
|---|---|---|---|---|---|---|---|---|
| NLANR | 10 | 26.6MB | 56.3KB | 25.9KB | 4 | 318KB | 132KB | 4 |
| WorldCup | 30 | 426KB | 31.0KB | 22.2KB | 4 | 96.3KB | 80.0KB | 4 |
| DEC-64 | 64 | 7.38MB | 1.69MB | 160KB | 12 | 4.61MB | 359KB | 4 |
| DEC-128 | 128 | 14.9MB | 7.19MB | 419KB | 14 | 24.6MB | 1.18MB | 6 |
| NLANR-203 | 203 | 44.3MB | 22.2MB | 1.20MB | 8 | 143MB | 4.24MB | 6 |
| UCB-512 | 512 | 78.0MB | 423MB | 16.1MB | 34 | 1.47GB | 31.2MB | 14 |

Table 2: Summary of data sets. The gzip'ed size of data series is also the byte count of the naive algorithm; "# of RTs" stands for number of round trips. Note that in the cases of DEC-128, NLANR-203 and UCB-512, TA consumes more bandwidth than the naive algorithm in terms of unicast bytes, since the central manager sends a long list of object URLs to all nodes for lookup.

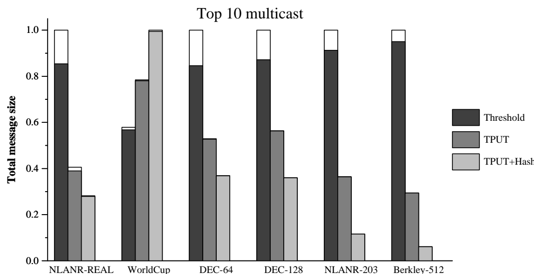


Figure 2: Broadcast bytes of the algorithms for finding the top 10 objects.

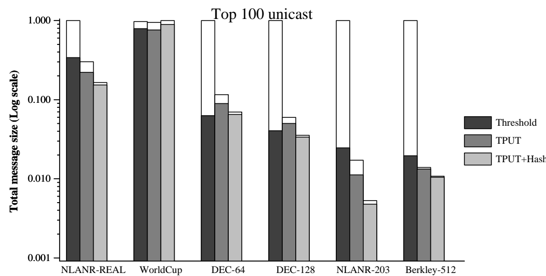


Figure 3: Uni-cast bytes of the algorithms for finding the top 100 objects. *Note that the y-axis is in log scale.*

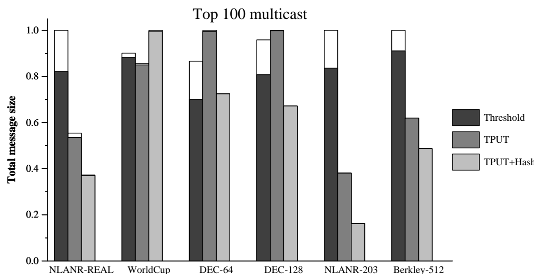


Figure 4: Broadcast bytes of the algorithms for finding the top 100 objects.

## 3.2 Performance Results

Performance comparison of TA, TPUT, and TPUT with hash compression are shown in Figures 1 to 4. We present the results on $k$ highest byte-count URLs for DEC-64 and DEC-128 traces, and the results on $k$ most referenced URLs on other traces. Results of other combinations are similar.

Figure 1 shows the uni-cast bytes of the algorithms for selecting the top 10 objects (i.e. $k = 10$), while Figure 2 shows the broadcast bytes. Figure 3 and Figure 4 are the corresponding results for selecting the top 100 objects (i.e. $k = 100$). The results are normalized against the algorithm with the highest cost. The shaded region in each bar represents the number of bytes sent by the nodes to the manager, while the empty region represents the number of bytes sent by the manager to the nodes.

As the number of nodes increases, the advantage of TPUT over TA in the unicast case magnifies. The main reason is that the number of objects looked up in all nodes is much lower in TPUT than in TA. For example, in the case of DEC-128 traces with $k = 100$, TA looks up a total of 26680 URLs in the 128 nodes, whereas TPUT only looks up a total of 250 URLs. TPUT is clearly very effective at reducing the set $S$.

Though not shown here, we experimented with different $\alpha$ values. Larger $\alpha$ tends to increase the size of candidate set $S$ but reduces the number of items that nodes send to the central manager, while smaller $\alpha$ has the opposite effect. Optimal $\alpha$ varies by traces, oscillating between 0.3 and 0.8. Overall, $\alpha = 0.5$ appears to be a good default choice.

In the case of Worldcup-30 at $k = 10$, TA performs slightly better than TPUT. The reason is that in the WorldCup case, the web servers are very well load-balanced, which means that object rankings among all web servers are very similar, a perfect scenario for TA.

The comparison of broadcast bytes is mostly determined by how "far down the lists" TA and TPUT stop at, since broadcast bytes are dominated by the traffic from the nodes to the central manager. The stopping value of TA is always strictly higher than the stopping value of TPUT. However, depending on whether the values at the same positions across the lists are similar or very different, either TA or TPUT might win in this case. The results show that in many traces TPUT outperforms TA, but in a few cases TA outperforms TPUT slightly. Considering the fact that the number of round-trips of TA is unpredictable, we would still recommend TPUT as the top-$k$ algorithm in the case where a broadcast mechanism is available.

The effect of hash array compression is not uniform across the traces. On NLANR and UCB data sets it reduces the bandwidth consumption by a factor of 2 to 4, but on the Worldcup traces it increases the bandwidth consumption by about 25%. The main reason is that TPUT already uses gzip compression, and gzip can do a very good job at reducing the number of bytes needed to represent an object's name, making the hash array compression somewhat redundant.

The results show clearly that TPUT should be the default top-$k$ algorithm for large-scale networks. Hash array compression can be used when the object names are too long.

# 4 Instance-Optimality of TPUT

As defined in [FLN01], instance optimality is a measure of how close an algorithm is to the optimal algorithm in the worst case. Let $A$ denote the class of all deterministic algorithms, and let $D$ denote the class of data series that we are interested in. For any algorithm $a \in A$, and any data series $d \in D$, we use $cost(a, d)$ to denote the cost of running $a$ over $d$. Then an algorithm $R$ is *instance optimal over $A$ and $D$* if $R \in A$ and there exist two constants $C_1$ and $C_2$ such that for every $a \in A$ and $d \in D$:

$$cost(R, d) \leq C_1 * cost(a, d) + C_2$$

The constant $C_1$ is called the optimality ratio of $R$.

Unfortunately, TPUT is not instance-optimal over all possible data series. If a data series has N objects with a fixed value that is just over $T$, TPUT will send all N objects to the central manager, while a more adaptive algorithm might avoid sending all N objects. Since N can go to infinity, TPUT is not instance optimal.

However, though TPUT is not instance optimal over all data series, nor can any algorithm that terminates in a fixed number of round trips regardless of inputs. Hence, we need to incorporate some descriptions of the data series in optimality ratio analysis. Below, we first define a class of algorithms called "fixed round-trip algorithms", then introduce a concept called "log-log slope function" to characterize data series.

## 4.1 Fixed Round-Trip Algorithms

We study algorithms that calculate top-$k$ queries in the following fashion. The algorithm is a series of round-trip communications between the central manager and the nodes, at the conclusion of which the central manager has the identities of the top-$k$ objects and their values. At each step (i.e. a round-trip communication), the central manager sends information to each node, and each node uses a certain criterion to select items on the list to sent to the central manager. The selection criterion is one of the following:

- *by position*: at node $i$, all items from positions 1 through $h_i$ ($h_i$ is specified by the manager) are sent to the central manager;
- *by name*: if the name of an object falls in a set of object names sent by the central manager, then the object and its value are sent to the central manager;
- *by value*: at node $i$, all objects with values higher than $t_i$ (specified by the manager) are sent to the central manager;

The cost of the algorithm is measured in bandwidth consumption, which is modeled as "units of communication", where each unit is either an $\langle object, value \rangle$ pair sent by a node to the manager, or an object name in a list sent by the manager to the nodes.

We call these algorithms *generic* top-$k$ algorithms because they do not rely on any properties in the names of objects or any properties in the value distributions of the list, and they do not require the nodes to perform complicated operations when deciding what to send to the central manager.

We say that a generic algorithm is a *fixed round-trip* algorithm if it never incurs more than $c$ number of round trips where $c$ is a constant independent of $k$, the number of node $m$, and the data series. The threshold algorithm is not a fixed round trip algorithm, but the TPUT algorithm is.

Intuitively, if an algorithm is fixed round-trip, then it must use the value criteria in at least one step,

otherwise it won't be able to correctly find the top $k$ objects. This is because the "by name" step does not let the central manager see any new object, and the "by position" step does not show the manager objects at lower positions that could be in the true top-$k$ set. The following theorem formalizes the argument.

**Theorem 4.1.** *Any fixed round trip algorithm which correctly finds the top-$k$ object for all data series and which does not require seeing all data in all nodes must include a "by value" criterion in at least one of its steps.*

*Proof.* Please see appendix. □

Unfortunately, use of the "by value" criteria means that a fixed round-trip algorithm cannot be instance optimal over all possible data series. It's easy to see why; a data series may have an arbitrarily large number of objects having a particular value $V$ that satisfies the "by value" criterion. The following theorem formalizes the argument.

**Theorem 4.2.** *Let $A$ be the class of all deterministic algorithms that correctly finds the top-$k$ objects in all data series. Let $D$ be the class of all possible data series. Then no fixed round-trip algorithm can be instance optimal in $A$ over $D$.*

*Proof.* Details of the proof are in Appendix. □

Hence, the general notion of instance optimality ratio cannot be used to characterize fixed round-trip algorithms. However, we can still analyze optimality ratios of these algorithms by incorporating a characterization of the data series themselves. The concept that we introduce is called *log-log slope function*, described in the next section.

## 4.2 Optimality Ratio of TPUT

We introduce the *log-log slope function* of a data series. In a sorted list, we call the value of the object at the $i$'th position in the list "the value of the $i$'th position", denoted as $w(i)$. Note that $w(i) \geq w(j)$ if $i \leq j$, since the list is reverse sorted.

We say that a data series has a log-log slope function $C(n)$ if, for all $i$ where $i \leq k$, $w(C(n) * i) < w(i)/n$, and $C(n)$ is the smallest value satisfying this criteria ($n$ is an integer here). In other words, in order to achieve a factor of $n$ reduction in the value, one has to go down a factor of $C(n)$ deeper down the list. For example, data series that follow Zipf-like distribution, that is, $w(i) = O(1/i^{\beta})$, has a log-log slope function of $C(n) = n^{1/\beta}$. Clearly, $C(n)$ is a non-decreasing function. The log-log slope function bounds the occurrence of a long "plateau" in the values in the sorted list, and it only needs to apply to the top $k$ ranked object as far as the algorithm is concerned.

TPUT is instance optimal over data series that satisfy log-log slope function $C(n)$. Put it differently, the optimality ratio of TPUT can be characterized by $C(n)$ of the data series. Before we prove that, we establish two properties of TPUT.

**Lemma 4.3.** *Phase-1 bottom, $\tau_1$, is at least $1/m$ of the true bottom $\tau$.*

*Proof.* After the central manager receives the top $k$ objects from every node, it calculates partial sums for these objects. Let's call the collection of these objects $S_1$. The central manager selects objects with the top $k$ partial sums, and sets $\tau_1$ to be the min of the top $k$ partial sums. Therefore, for an arbitrary subset of $k$ objects in $S_1$, the min of their partial sums is $\leq \tau_1$. In other words, $\tau_1$ is no less than the bottom partial sum of any subset of $k$ objects in $S_1$.

Now, let's sort all the $m * k$ pairs of $\langle object, value \rangle$ received in Phase 1 by value. Then go down the sorted list, and find the first value, $t$, that belongs to the $k$'th object that has been seen in going down the list. Clearly, the set of $k$ objects that are seen in going down the list has a bottom value that is at least $t$. Hence, $t \leq \tau_1$.

Now, for any object to have a value that is higher than $m * t$, it must have a value in a node that is higher than $t$. Since the number of objects that have values higher than $t$ is at most $k - 1$, it follows that one can't find $k$ objects whose values are higher than $m * t$, which is saying in another way that the true bottom is at most $m*t$. Since $t \leq \tau_1$, the true bottom is at most $m * \tau_1$. □

It's easy to see that the above bound is tight by constructing an example where all $m * k$ objects are different.

**Lemma 4.4.** *Phase-2 bottom, $\tau_2$, is at least $\tau * m/(2m - 1)$, where $\tau$ is the true bottom.*

*Proof.* In phase 2, the difference between the partial sum of an object and its true sum is at most $T*(m-1) = (\tau_1/m) * (m - 1)$, since at whichever node that has not reported the object, its value in that node is at most $T$. Hence, the partial sums for the true top-$k$ set of objects are at least $\tau - \tau_1 * ((m - 1)/m)$. Since $\tau_2$ is at least the bottom partial sum of any set of $k$ objects, we have $\tau_2 \geq (\tau - \tau_1 * ((m - 1)/m)$.

Note also that $\tau_2 \geq \tau_1$, since the partial sum of any object in phase 2 is at least its partial sum in phase 1. Hence, $\tau_2 \geq max(\tau_1, \tau - \tau_1 * (m - 1)/m) \geq (\tau * m/(2m - 1))$.

□

By similar argument we have the following corollary.

**Corollary 4.5.** *If a pruning parameter $\alpha < 1$ is used, i.e. $T = (\tau_1/m) * \alpha$, then $\tau_2 > \tau * 1/(1+\alpha)$.*

Lemma 4.4 shows that by the end of phase 2, the central manager has a lower bound that is within a factor of 2 of the true bottom. Thus, in the case where the phase-1 bottom is too low and a lot of objects are sent to the central manager, the phase-2 bottom is much closer to the true bottom and can weed out many objects. Note that this property is true because the threshold value used in all nodes is the same, indicating the importance of using a uniform threshold.

Now, we are ready to prove the optimality ratio of TPUT.

**Theorem 4.6.** *Let $D$ be the class of all data series that have log-log slope function of $C(n)$. Let $A$ be the class of all deterministic algorithms that correctly finds the top $k$ answers for every data series in $D$. Then the basic TPUT algorithm (without pruning parameter $\alpha$) is instance-optimal over $A$ and $D$, with optimality ratio: $(m-1) * min(C(2m), C(m) * k) + min(C(m^2), C(m) * k)$.*

*Proof.* Assume algorithm $a \in A$, and assume a set of $m$ data series $d \in D$. For each data series $d_i \in d$, assume that $a$ stops at position $b_i$ in $d_i$, that is, $a$ sees the top $b_i$ objects in $d_i$, but does not see the object at position $b_i + 1$. As a result the object at position $b_i + 1$ does not belong to the true top-$k$ set. Then at node $i$, the value of position $b_i$, $w(b_i)$, satisfies $w(b_i) \leq \tau$, since otherwise the object at position $b_i+1$ might have a value higher than $\tau$ and $a$ cannot stop at position $b_i$.

Since the threshold $T = \tau_1/m$ and $\tau_1 \geq \tau/m$ as per Lemma 4.3, $T \geq \tau/m^2$. Therefore, $T \geq w(b_i)/m^2$. Based on the definition of the log-log slope function, objects at positions later than $C(m^2) * b_i$ have values $< w(b_i)/m^2$. Hence, TPUT stops at no later than $C(m^2) * b_i$ at data series $d_i$.

In addition, note that in node $i$, the value of the $k$'th ranked object in that node is $\leq \tau_1$. Since $T = \tau_1/m$, TPUT will stop at no deeper than position $C(m) * k$ in data series $d_i$.

Hence, the number of objects that TPUT sees in phase 2 is at most $min(C(m^2) * b_i, C(m) * k)$ which is at most $min(C(m^2), C(m) * k) * b_i$ since $b_i \geq 1$.

In phase 3, where the central manager asks for values of a set of objects, each object will have at most $m-1$ nodes sending the information to the central manager. Only if an object has a value $\geq \tau_2/m$ in at least one node can it belong to the set. Since at each node $d_i$, the value at position $b_i$ (where algorithm $a$ stops) is at most the true bottom $\tau$, and $\tau_2 > \tau/2$ as per Lemma 4.4, we know that at most $C(2m) * b_i$ objects have values higher than $\tau_2/m$. Hence, the total number of objects sent by the central manager in phase 2 is at most the total number of objects seen by algorithm $a$ times $C(2m)$. This number is also at most the number of objects seen by algorithm $a$ times $C(m) * k$ as per the analysis above. Therefore, the amount of information received by the central manager is at most $(m-1) * min(C(2m), C(m) * k)$ times the amount of information seen under algorithm $a$.

Combining the phase 2 and phase 3 analysis gives us the optimality ratio.

Furthermore, the ratio is tight because one can construct the following example that incurs the ratio. In the example, k=1. Every node with the exception of node 1 has a top object that has value 1 and does not appear in any other node. Node 1's top object appears in other nodes at position 2 with value 1. All other objects in the nodes do not appear in more than one node. The optimal algorithm would just incur $2m$ communication units to find the top object, while the basic TPUT algorithm would fetch $C(m)$ objects from each node and look up $m * C(m)$ objects in $m-1$ nodes, leading to the optimality ratio.

□

How close is TPUT to an optimal fixed round-trip algorithm? Below, we give a weak lower bound on the optimality ratio of any fixed round-trip algorithm.

We define "unit data series" as the following class of data series. Each node has a list of objects of value 1, and the objects, except for the very last one, do not appear in any other node. The last object in all nodes' list is the same and has value 1 in all nodes. We say that a fixed round trip algorithm has depth $p$ for unit data series, if, before the "by value" step, the shallowest position it has gone down in any list is position $p$. The depth of any fixed round trip algorithm on the unit data series is finite because the algorithm can only run a fixed number of steps using the "by position" criterion.

**Theorem 4.7.** *Let $A$ be the class of all deterministic algorithms. Let $D$ be the class of all data series that have a log-log slope function of $C$. Then for any fixed round trip algorithm $B$, its optimality ratio cannot be lower than $C(m) * m/(p+1+2m)$, where $p$ is the depth of $B$ on the unit data series.*

*Proof.* For any algorithm $B$, an adversary can construct a data series that is a variation of the unit data series. $B$'s cost on this data series is $C(m) * m$, and

there exist an algorithm that would only incur cost $p+1+2m$. Details of the construction can be found in the appendix. □

If an algorithm uses only three round trips, its first step must be a "by position" step and its depth on the unit data series $p$ is the minimum of the $h_i$'s that it uses on the nodes. Since the algorithm gathers all $h_i$ objects from node $i$ even if the top object is the same on all nodes and is the top-1 object, the algorithm has an optimality ratio of at least $p$. Hence, all three round trip algorithms have optimality ratios that are $\geq max(p, C(m)*m/(p+1+2m)) \geq \sqrt{C(m)*m+(m+1/2)^2}-(m+1/2)$.

**Corollary 4.8.** *Any three round trip algorithm has an optimality ratio of at least* $\sqrt{C(m)*m+(m+1/2)^2}-(m+1/2)$ *on data series that satisfy the log-log slope function* $C$*.*

Though the above lower bound is quite weak, it does show that the dependence on $C(m)$ is inherent to all fixed round-trip algorithms.

### 4.3 Effects of the Parameter $\alpha$

The pruning parameter $\alpha < 1$ has a surprising impact on the size of the candidate set $S$. Without it, if $\tau_1 == \tau_2$, then no objects can be pruned away. More fundamentally, for any object in $S$, $\alpha$ introduces a coupling between its value and the number of nodes that reports it in phase 2. Intuitively, if an object appears in a few nodes and still "makes the cut", then its value must be high in those nodes; if an object has low values but "makes the cut", then it must appear in many nodes. Below, we formalize the argument, and show the power of any $\alpha < 1$ in the case of Zipf distribution.

To analyze the size of $S$ under a particular $\alpha$, we construct the following 2-D matrix. Each row in the matrix is the name of an object in $S$. Each column is a node. We set $T' = (\tau_2/m)*\alpha$ (in the case of $\tau_2 = \tau_1$, $T' = T$). The entry $\langle o, i\rangle$ is 1 if object $o$ appears in the list sent by node $i$ and $v_i(o) \geq T'$, and 0 otherwise. Note that each row has at least one entry that is 1.

Clearly, the sum of all entries in column $i$ is less than or equal to the number of objects in node $i$ with values $\geq T'$. Since $T' \geq \tau*\alpha/(m*(1+\alpha))$, by analysis in the previous section, the sum of all entries in column $i$ is $\leq C(m*(1+\alpha)/\alpha)*b_i$. Thus, if we sum all the entries in this matrix by column, then the sum is $\leq C(m*(1+\alpha)/\alpha)*\Sigma(b_i)$.

Let $x_l$ denote the number of objects in the matrix that appears in exactly $l$ columns. Then if we sum all the entries by rows, the sum $\Sigma(x_l*l)$ equals to the sum of all entries by columns and hence $\Sigma(x_l*l) \leq C(m*(1+\alpha)/\alpha)*\Sigma(b_i)$.

Now, if an object $p$ appears in less than $l$ columns and "makes the cut" (i.e. the upper bound of its true sum is over $\tau_2$), its average value in those nodes, $R$, must satisfy $(R*l)+T*(m-l) \geq \tau_2$ (note also that $R \geq T$). Since $T \leq (\tau_2/m)*\alpha$, $(R*l) \geq \tau_2*(1-((m-l)/m*\alpha)) \geq \tau_2*(1-\alpha)$. Since $\tau_2 \geq \tau/(1+\alpha)$, $R \geq \tau*(1-\alpha)/(1+\alpha)*1/l$. Let node $i$ be the node where $p$'s value is higher than its average $R$, then it's clear that $p$ must appear in node $i$'s sorted list at a position no deeper than $b_i*C(l*(1+\alpha)/(1-\alpha))$. Let $\beta = (1+\alpha)/(1-\alpha)$. If we count each such object exactly once by choosing a node where the object's value is higher than or equal to its average value, then we have $x_1+x_2+\ldots+x_l \leq C(l*\beta)*\Sigma(b_i)$.

The number of objects in $S$ is simply $\Sigma(x_l)$ where $l = 1, \ldots, m$. We have two constraints:

1. $\Sigma(x_l*l) \leq C(m*(1+\alpha)/\alpha)*\Sigma(b_i)$;
2. $x_1+x_2+\ldots+x_l \leq C(l*\beta)*\Sigma(b_i)$ for each $l$;

where $\beta = (1+\alpha)/(1-\alpha)$.

The maximum value of $\Sigma(x_l)$ occurs when these exists a value $l_0$ such that for each $l \leq l_0$, $x_l$ is the maximum allowed under constraint 2, and for each $l > l_0$, $x_l = 0$. In other words, $x_1 = C(\beta)*\Sigma(b_i)$, $x_2 = (C(2*\beta))-C(\beta))*\Sigma(b_i)$, $x_3 = (C(3*\beta)-C(2*\beta))*\Sigma(b_i)$, and in general $x_i = (C(i*\beta)-C((i-1)*\beta))*\Sigma(b_i)$. The value $l_0$ is simply the maximum value such that $l_0*C(l_0*\beta)-C((l_0-1)*\beta)-C((l_0-2)*\beta)-\ldots-C(\beta) \leq C((1+\alpha)*m*1/\alpha)$. The size of $S$ (i.e. the total number of "random lookup" objects) is $C(l_0*\beta)*\Sigma(b_i)$. $l_0$ is determined by the log-log slope function. Hence, we can refine the optimality ratio below.

**Theorem 4.9.** *If the TPUT algorithm uses the parameter* $\alpha$ *(*$0 < \alpha \leq 1$*) to improve the pruning power, then the optimality ratio is at most* $(m-1)*C(l_0*(1+\alpha)/(1-\alpha))+min(C(m^2*1/\alpha), C(m*1/\alpha)*k)$*, where* $l_0$ *is defined above.*

The relationship between $l_0$ and $m$ depends on the log-log slope function $C$. In the case of Zipf distribution, where $C(l) = l$, we have $l_0^2+l_0 \leq m*(1-\alpha)/\alpha$. In other words, $l_0$ is approximately $\sqrt{m*(1-\alpha)/\alpha}$.

**Corollary 4.10.** *TPUT with pruning parameter* $\alpha < 1$ *has an optimality ratio that is* $O(m*\sqrt{m})$[1] *for Zipf distribution, regardless of the value of* $\alpha$*.*

The analysis above shows the impact of $\alpha$ on the size of $S$. The parameter $\alpha$ also has another effect, that is, it reduces the threshold and increases the

[1] We suspect that this ratio is tight within a constant factor for Zipf distribution, but we are yet to construct a detailed example that exhibits this ratio.

number of objects that each node sends to the central manager. This is reflected in the second additive term in the optimality ratio in Theorem 4.9. Hence, the optimal $\alpha$ that minimizes network traffic depends on both the log-log slope function and the number of nodes $m$. In our experience, we found that $\alpha = 0.5$ appears to work well across a range of systems and we use it as the default value.

# 5 Extending TPUT to Hierarchical and P2P Networks

TPUT can be easily extended to hierarchical and peer-to-peer networks. The algorithm would still operate in three phases, which are *lower-bound estimation*, *pruning*, and *final lookup*, but the operations in each phase vary according to the network topology. Below, we discuss how TPUT might operate in a two-level hierarchical network and the associate design choices.

Assume that the network is a two-level tree hierarchy, with the central manager talking to $m$ intermediate nodes, and each intermediate node $i$ talking to $n_i$ leaf nodes. For simplicity we assume that only the leaf nodes have data. The top-$k$ query is then aggregating values over all leaf nodes.

For phase 1, there are numerous ways to obtain a lower bound. One approach is for all leaf nodes to send their top-$k$ elements to the central manager. The lower bound obtained this way could be a factor of $1/\Sigma n_i$ less than the true bottom. Another approach is for each of the intermediate node to initiate its own top-$k$ query over its children and then send the results to the central manager. The lower bound obtained this way would be no less than $1/m$ of the true bottom, but more communications are initiated. Yet a third approach would be for the central manager to take whatever data it got and make a guess on the lower bound. If it guesses too high it will find out at end of Phase 3, and it can adjust the estimate and re-run the algorithm.

For phase 2, the "by value" query from the central manager is easily decomposable. The central manager sets the threshold $T = (\tau_1/m) * \alpha$, and sends the threshold to intermediate nodes. Each intermediate node $i$ then sets a next-level threshold $T' = (T/n_i) * \alpha'$ and issues a new "by value" query to all its children. After receiving all replies, the intermediate node can either go through a round of phase 3 lookup of its own to determine exactly the answers to send to the central manager, or simply estimate the set through upper bound calculations and send the estimated superset to the central manager.

For phase 3, the "by name" query from the central manager is propagated from the intermediate nodes to the leaf nodes, except that some of the lookups can be eliminated at the intermediate node. For each object that is looked up, the central manager attaches its current partial sum, upper bound on sums from all other nodes, and $\tau_2$. If the intermediate node has the value of the object or has an upper bound estimate on the object such that the object's value cannot be higher than $\tau_2$, then the object can be eliminated from the lookup.

TPUT running over multi-level hierarchies is a recursive extension of the two-level hierarchy operation. Due to space limitation we omit the details here.

To calculate top-$k$ query over peer-to-peer networks, one can first establish a min-depth broadcast tree over the network, then run the algorithm over the broadcast tree. There are many ways to establish the min-depth broadcast tree, including flooding in unstructured peer-to-peer networks or utilizing the inherent network structures in structured peer-to-peer networks. Detailed investigations of TPUT on P2P networks are part of our future work.

# 6 Related Work

The database research community have long studied the issue of efficient processing of top-$k$ queries [NR99, Fag99, UBK00, BGM02, FLN01, BO03], since they are prevalent in handling heterogeneous data such as multimedia data. In particular, the threshold algorithm was discovered independently by (at least) three groups [NR99, UBK00, FLN01]. One big difference between our study and these studies is how big $m$ is. In the database systems, $m$ is the number of databases that the query is accessing, and often $m$ is small. In contrast, we are interested in large scale networks and $m$ is large.

Our work benefited greatly from the seminal paper on this subject by Fagin, Lotem and Noar [FLN01]. In particular, the concept of instance optimality is from that study. The pruning technique used in TPUT is similar to the upper-bound/lower-bound technique used in the Quick-Combine algorithm in [UBK00] and the "Combined Algorithm" (CA) in [Fag99]. However, those algorithms do not apply in our environment as they require too many round trip communications. Furthermore, the use of Uniform Threshold and the pruning parameter $\alpha < 1$ are new in our algorithm.

The use of threshold in TPUT is somewhat similar to the use of range queries in the top-$k$ selection

algorithm in [CG99], particularly the "no-restarts" strategy in setting search score. However, [CG99] does not consider the cost of "looking up" objects in databases, hence has a different cost model from ours. The algorithm in [CG99] uses histograms heavily. We believe that per-node histograms would be of very limited use in our environments, since the histogram distribution of values in one node says nothing about an object's values in other nodes. We do note that efficient calculation of aggregate histogram distribution would be an interesting research question for distributed networks.

The distributed top-$k$ monitoring study by Bobcock and Olston [BO03] looks at a network environment that is similar to ours. However, their study is focused on monitoring whether the set of top $k$ objects have changed after an initial answer has been obtained, and they simply use the threshold algorithm to obtain the initial answer. We are interested in algorithms that can obtain the initial answer efficiently. The reason is that in our target environments the query is asked hourly or daily. The intervals between the queries are typically long enough that the top-$k$ objects have changed completely, and it's more efficient to serve the queries on demand.

Our use of hash array compression is similar to techniques using hash array counters in the "iceberg" study [FSGM+98], the router traffic measurement study [EV01] and the spectral bloom filter study [CM03]. However, those studies are interested in finding out the set of "iceberg" objects, i.e. objects whose values account for 90% of total value, in a non-distributed environment. In contrast, in our study the top-$k$ objects might not be "icebergs" and our algorithm runs in a distributed environment. As a result, while hash array counters are essential in those algorithm, they only provide a constant factor speedup in our algorithm.

# 7 Conclusions and Future Work

In this paper we study efficient top-$k$ algorithms for distributed networks and present the three-phase uniform threshold (TPUT) algorithm. TPUT takes only three round trips over a star network, and significantly out-performs existing algorithms such as the Threshold Algorithm. It is instance-optimal over common data distributions. Trace-driven studies show that on large networks, the traffic of TPUT can be two orders of magnitude less than those of existing algorithms.

Our future work lies in three areas. First, we plan to investigate the various design choices in top-$k$ algorithms for multi-level hierarchical networks and peer-to-peer networks. Second, we plan to provide a library of top-$k$ calculations for distributed computing infrastructures such as the PlanetLab. Lastly, we plan to incorporate top-$k$ query calculation mechanisms in predictive replication systems in a content distribution network.

# References


[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[BGM02] Nicolas Bruno, Luis Gravano, and Amelie Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

[BO03] Brian Bobcock and Chris Olston. Distributed top-$k$ monitoring. In *SIGMOD*, 2003.

[CG99] Surajit Chaudhuri and Luis Gravano. Evaluating top-k selection queries. In *VLDB'99*, pages 397–410, 1999.

[CM03] Saar Cohen and Yossi Matias. Spectral bloom filters. In *SIGMOD/PODS*, 2003.

[Cor96] Digital Equipment Corporation. Anonymized web proxy traces. Technical report, http://ftp.digital.com/pub/Digital/traces/proxy/webtra 1996.

[EV01] C. Estan and G. Varghese. New directions in traffic measurement and accounting, 2001.

[Fag99] R. Fagin. Combining fuzzy information from multiple systems. In *J. Comput. System Sci.*, pages 58:83–99, 1999.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Symposium on Principles of Database Systems*, 2001.

[FSGM+98] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing

iceberg queries efficiently. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 299–310, 24–27 1998.

[IRC03] IRCache. Traces from the ircache system. Technical Report http://www.ircache.net, National Laboratory for Applied Network Research, 2003.

[NR99] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29, 1999.

[UBK00] U.Guntzer, W-T. Balke, and W. Kiessling. Optimizing multi-feature queries in image databases. In *Proc. 26th Very Large Databases (VLDB) Conference*, pages 419–428, 2000.

# A Detailed Proofs

We give detailed proofs for several theorems in Section 4.1 here.

**Theorem A.1.** *Any fixed round trip algorithm which correctly finds the top-k object for all data series and which does not require seeing all data in all nodes must include a "by value" criterion in at least one of its steps.*

*Proof.* For any algorithm that does not use the "by value" criterion and yet finishes within a constant number of round trips, an adversary can construct a data series for which the algorithm cannot identify the correct top-$k$ objects.

The adversary operates as the following. As the algorithm progresses, at each "by position" step, the adversary creates objects whose values are 1 and whose names do not appear in any other node, and at each "by name" step, the adversary sets the object's value to be 0 unless the object is already appearing in the node. When the algorithm ends, assume that for each node $i$, it has seen objects up to position $h_i$, the adversary now constructs $k$ objects that appear at positions $h_i+1,\dots,h_i+k$ at each node $i$, all with value 1, and these $k$ objects are the top-$k$ objects, but the central manager hasn't seen them. Hence the algorithm does not identify the correct answers. □

**Theorem A.2.** *Let $A$ be the class of all deterministic algorithms that correctly finds the top-k objects in all data series. Let $D$ be the class of all possible data series. Then no fixed round trip algorithm can be instance optimal in $A$ over $D$.*

*Proof.* Without lose of generality suppose $k = 1$.

Let $B$ be a fixed round trip algorithm. An adversary constructs the following data series based on the action that $B$ takes at each step. If the step is "by position", then the adversary returns a collection of objects that do not appear in any other node and that have values of 1. If the step is "by name", then the adversary set the values to be 0.

Now, if it's "by value", let's say by values that are $\geq C$, then pick a $C'$ such that $1 > C' > C$. Now, after the bottom positions in all data series, put an object $o$ with value 1 at each node. Then put $N$ objects of value $C'$ in each data series. It's clear that an algorithm that takes the existing step and then does one more look into the data series will find the object $o$ and terminate, but the algorithm $B$ will fetch all $N * m$ data pairs. Since $N$ can go to infinite, $B$ cannot be instance optimal.

□

**Theorem A.3.** *Let $A$ be the class of all deterministic algorithms. Let $D$ be the class of all data series that have a log-log slope function of $C$. Then for any fixed round trip algorithm $B$, its optimality ratio cannot be lower than $C(m) * m/(p + 1 + 2m)$, where $p$ is the depth of $B$ on the unit data series.*

*Proof.* Without loss of generality we assume $k = 1$. The proof can be extended to $k > 1$.

For any algorithm $B$, an adversary constructs the following variation on the unit data series. The data series is a unit data series until the depth $C(m)$, at which point the values at the bottom position in each list is $1/m$. After that position the list is again an infinite list of non-overlapping objects with value $1/m^2$.

The adversary now changes the data series according to how $B$ operates. After $B$ has gone through its "by position" and "by name" steps and comes to its first "by value" step, it must have stopped at position $p$ in one of the nodes, say node $j$. The adversaries then change the objects at all $C(m)$ positions in all other nodes to be the same object as the object at position $p+1$ in node $j$. Thus, $B$ has not identified the top-1 object and must continue onto the "by-value" step.

However, an algorithm that goes down to position $p+1$ in node $j$ and position 1 in all other nodes, then does a lookup of the object at position $p+1$ in all other nodes, will immediately identify the top object. In other words, there exists an algorithm that sends $m+p+m$ $\langle object, value\rangle$ pairs to the central manager.

Now, $B$'s "by value" step will have a value $b_i <= 1$ for each node $i$, at which point it will pick up all $C(m) * m$ $\langle object, value\rangle$ pairs. Hence, $B$ incurs at

least $C(m) * m$ units of communication. Since there exists an algorithm that incurs $p + 1 + 2m$ units of communication on the data set, the optimality ratio is at least $C(m) * m/(p + 1 + 2m)$.

□