Finding the K Highest-Ranked Answers in a Distributed Network

Demetrios Zeinalipour-Yazti [a,*] Zografoula Vagena [b]
Vana Kalogeraki [c] Dimitrios Gunopulos [c] Vassilis J. Tsotras [c]
Michail Vlachos [d] Nick Koudas [e] Divesh Srivastava [f]

[a] *University of Cyprus, Nicosia, Cyprus*
[b] *Microsoft Research Cambridge, Cambridge, UK*
[c] *University of California - Riverside, Riverside, CA, USA*
[d] *IBM Research Zurich, Rueschlikon, Switzerland*
[e] *University of Toronto, Toronto, ON, Canada*
[f] *AT&T Research Labs, Florham Park, NJ, USA*

Abstract

In this paper we present an algorithm for finding the k highest-ranked (or Top-k) answers in a distributed network. A Top-K query returns the subset of most relevant answers, in place of all answers, for two reasons: i) to minimize the cost metric that is associated with the retrieval of all answers; and ii) to improve the recall and the precision of the answer-set, such that the user is not overwhelmed with irrelevant results. Our study focuses on multi-hop distributed networks in which the data is accessible by traversing a network of nodes. Such a setting captures very well the computation framework of emerging Sensor Networks, Peer-to-Peer Networks and Vehicular Networks. We present the *Threshold Join Algorithm (TJA)*, an efficient algorithm that utilizes a non-uniform threshold on the queried attribute in order to minimize the transfer of data when a query is executed. Additionally, *TJA* resolves queries in the network rather than in a centralized fashion which further minimizes the consumption of bandwidth and delay. We performed an extensive experimental evaluation of our algorithm using a real testbed of 75 workstations along with a trace-driven experimental methodology. Our results indicate that *TJA* requires an order of magnitude less communication than the state-of-the-art, scales well with respect to the parameter k and the network topology.

Key words: Distributed Top-K Query Processing, P2P Networks, Sensor Networks

1 Introduction

The widespread deployment of networked computing devices and the emerging ubiquitous computing paradigm have brought a shift from traditional centralized computing infrastructures to interconnected networks of peers, sensors and vehicles. In such networks, nodes connect to a small non-uniform number of other nodes which in effect creates an overlay communication graph that is utilized for data sharing and query processing.

A typical characteristic of such networks is that nodes maintain or generate large amounts of data. For instance, a typical sensor device can sample environmental parameters, such as pressure, humidity and temperature at rates ranging from 2Hz to 500Hz [34]. Thus, the task of querying a collection of sensors is confronted with the fundamental challenge in distributed query processing: *"how to find the expected answer without observing all the distributed relations in their entirety"*. Instead of retrieving all the answers we focus on efficiently retrieving the k highest-ranked (or *Top-k*) answers and minimize at the same time the cost of evaluating the query.

In this paper we present the *Threshold Join Algorithm (TJA)*, which is an efficient Top-k query processing algorithm for distributed environments. An example of a Top-k query might be: *"Find the three moments on which we had the highest average temperature in the last month across all sensors deployed on a mountainside."* The objective of a Top-k query is to find the k highest ranked answers to a user defined similarity function. Our algorithm is designed for queries where the user is interested in having the k most relevant answers sporadically, rather than continuously. For example, biologists analyzing a forest [39] are usually interested in long-term monitoring that will allow them to understand changes and conditions over long periods of time. Therefore various sensors can record environmental readings from their surrounding environment to local storage, and then selectively transmit their tuples to a query processing node (*the sink*), when certain preconditions are met, or when the sensors receive a query over the radio [41].

Since the execution of a query in a distributed environment is typically associated with the transfer of data over an expensive wireless or wired commu-

* *Contact Author: dzeina@cs.ucy.ac.cy, Tel: +357-22-892755, Fax: +357-22-892701*

Email addresses: zeinalipour@ouc.ac.cy (Demetrios Zeinalipour-Yazti), zografv@microsoft.com (Zografoula Vagena), vana@cs.ucr.edu (Vana Kalogeraki), dg@cs.ucr.edu (Dimitrios Gunopulos), tsotras@cs.ucr.edu (Vassilis J. Tsotras), vlachos@us.ibm.com (Michail Vlachos), koudas@cs.toronto.edu (Nick Koudas), divesh@research.att.com (Divesh Srivastava).

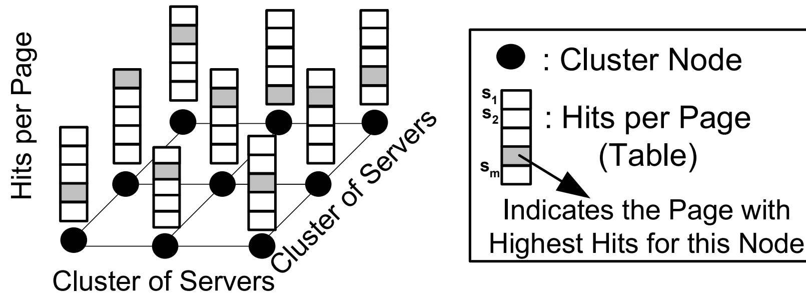

Fig. 1. **Distributed Top-k query processing in the context of a Content Distribution Network:** Given a set of n servers (cluster nodes), each recording locally the number of hits per page s_i $(i \leq m)$, we are interested in finding the k pages with the highest number of hits across all n servers. The objective is to complete this task without pulling together all the distributed logs.

nication medium, TJA's objective is to minimize this burden by transferring fewer readings to the sink and to execute in a fixed number of round trips. The former parameter is important because the minimization of the network utilization also reduces processing and energy consumption. The latter parameter on the other hand, is important because executing an algorithm in a fixed number of phase minimizes expensive round-trips over an expensive communication medium. Additionally, *TJA* resolves queries in the network rather than in a centralized fashion and that further minimizes the consumption of bandwidth and delay.

Distributed *Top-k* query processing is useful in an ever growing number of other domains, such as Content Distribution Networks, Internet Traffic Monitoring, Information Retrieval and Spam Detection Networks. Below, we describe their applicability in more detail for some of these settings:

Example 1 - Content Distribution Networks: Let $S = \{s_1, s_2, ..., s_m\}$ denote the m pages of a popular web site S (see Figure 1). For scalability and fault tolerance reasons, the m pages are replicated over a set of n servers (each server maintains on local disk the m web pages). Whenever a webpage s_i $(i \leq m)$ is accessed by some user, the redirector (router) forwards the user to one of the n servers. Let each server n_j $(j \leq n)$ record these visits in a local log. In many occasions the system administrators are interested in queries that refer to the complete set of distributed servers, such as: *"find the page with the highest number of hits across all servers"*. An answer to such a query would enable Administrators to reorganize the linkage structure of S in such a way that the most popular pages become available with the fewer number of user clicks. It is important to note that the router does not perform any application-layer handling of incoming TCP traffic. Thus, the router cannot simply count the incoming traffic of each server in order to define the answer to the above query.

Fig. 2. **Distributed Top-k query processing in the context of Internet Traffic Monitoring:** The illustration on the left presents two monitors (squares) that ping a number of landmarks (dots). The collected data is utilized to infer topological properties about the Internet. A Top-k query is to find the k landmarks (regions) with the highest score in some specified metric (loss, delay, etc). The figure on the right depicts the bipartite graph abstraction of this scenario.

Obviously, such a query can only be answered if the user first fetches all the distributed logs to a centralized repository and then combines these relations using some traditional centralized Top-k query processing algorithm [14,15]. This is fairly expensive as the number of web-servers and the number of web-pages might be very large. The techniques proposed in this paper alleviate this burden by transferring only the tuples that are most likely to be in the final result.

Example 2 - Internet Traffic Monitoring: In Internet Tomography [10], researchers infer topological properties about the Internet through indirect measurements. In this context, Content Distribution Networks, such as Akamai [1], and research organizations, such as *NLANR* [18], continually measure distances, such as latency or loss, from a very large number of *monitors* to a large set of representative landmarks, such as well chosen DNS servers. An illustration of this scenario is depicted in Figure 2. Such measurements allow scientists to understand traffic patterns, predict failures, map the structure the of the Internet on an ongoing basis and many others. Infrastructure operators and researchers are often confronted with queries that refer to all monitors, such as: *"find the region that has the highest distance (e.g., loss, latency) from all monitors in the last 15 minutes"*. To answer such queries, somebody must gather the pings from all monitors for the specified window of time. This is an endeavor task as the number of monitors and landmarks is usually very large.

The techniques discussed in this paper minimize the transfer of data, and thus the total time to find the expected answers. Our techniques additionally minimize computation, which leads to significant conservation of energy. This is particularly important when dealing with battery-powered wireless sensor devices [39–41]. Our techniques increase the available energy budget of a device, which can in effect be utilized for increasing the sampling frequency

and hence the fidelity of the measurements in reproducing the actual physical phenomena, or for just prolonging the lifetime of a device.

This paper builds upon our previous work in [40], in which we presented the design and initial results of the Threshold Join Algorithm (TJA) in a sensor network environment. In this paper we introduce several new improvements which are summarized as following:

- We present a study of the TJA algorithm using a real middleware system we developed on a network of 75 workstations. We additionally conduct the following studies: i) We compare the TJA algorithm to another three-phase algorithm, the TPUT [8] algorithm, and show that our algorithm has superior performance because it utilizes a non-uniform threshold rather than a uniform threshold; ii) We empirically assess the scalability of the parameter k which signifies the number of expected results to a query; and iii) We probe our algorithm on network topologies of different diameters. Our study provides solid experimental evidence about the performance and scalability characteristics of the TJA algorithm in a real system deployment;
- We provide a formal and detailed correctness proof of the Threshold Join Algorithm and its internal structures showing that it yields a correct answer in every instance;
- We provide an extensive overview of related work and a taxonomy of related algorithms based on four different dimensions: *number of phases*, *input scores* (exact, approximate), *output ranking* (exact, approximate) and *query type* (on-demand, continuous).

2 Problem Definition and Outline of Operation

In this section we will formalize our basic terminology upon which we will build the description of our algorithm. We will then outline the motivation behind the phases of the Threshold Join Algorithm.

2.1 Problem Definition

Let R be a relation with n attributes $\{s_1, s_2, \ldots, s_n\}$, each featuring m objects $\{o_1, o_2, \ldots, o_m\}$. The jth attribute of the ith object is denoted as o_{ij} (see Table 1). Also let $G(V, E)$ denote an undirected network graph that interconnects the n vertices in V using the edge set E. The edges in E, represent the connections between the vertices in V (the set of nodes). We assume that each vertex is connected to only d ($d << n$) other vertices (i.e., the average degree of the graph is d). For ease of exposition, assume that each node s_i

is mapped to the elements of the vertex set $V = \{v_1, v_2, \ldots, v_n\}$ using a 1:1 mapping function $f : s_i \rightarrow v_i, \forall i$. This assumption defines that each node only maintains local information (i.e., a single dimension in the n-dim space).

Consider $Q = (q_1, q_2, \ldots, q_n)$, a Top-k query with n attributes. Each attribute of Q refers to the corresponding attribute of an object and the query aims to find the k objects with the maximum value in the following scoring function:

$$Score(o_i) = \sum_{j=1}^{n} sim(q_j, o_{ij}) \quad (1)$$

where $sim(q_j, o_{ij})$, is a similarity function that evaluates the jth attribute of the query Q against the jth attribute of an object o_i and returns a value in the domain R^+ (a higher value denotes a higher similarity). Similarly to [13,7], we require that the scoring function is *monotone*. A function is monotone if $sim(q_j, o_{1j}) > sim(q_j, o_{2j})$ ($\forall j \in n$) then $Score(o_1) > Score(o_2)$.

For example, assume a collection of temporal climate objects from three sensors s_1, s_2 and s_3 at two time moments o_1:(s_1=100F, s_2=90F, s_3=80F) and o_2:(s_1= 100F, s_2=70F, s_3=80F). Furthermore, assume that the distance function $sim(q_j, o_{ij})$ represents a fraction of the maximum value on dimension j. Given that some function *max* calculates the highest temperature in a given dimension, the top-1 object to the query Q= (max(temp), max(temp), max(temp)), would be o_1 because Score(o_1)=3.0 (i.e., 1*1.0 + 1*1.0 + 1*1.0) and Score(o_2)= 2.77 (i.e., 1*1.0 + 1*0.77 + 1*1.0).

2.2 Systems Parameters

As we mentioned earlier, a Top-K query returns the subset of most relevant answers to the query according to a cost metric. Since such a cost metric can come in different forms, depending on the host environment, in this work we define it based on the number of *Network Bytes* (i.e., the sum of Bytes of all nodes that reply to the Top-K query) and the number of *Network Messages* utilized to compute a Top-k answer (i.e., the sum of messages of all nodes that reply to the query). The difference between these two parameters originates from the fact that in networking protocols messages might have variable length sizes. Consequently, a single large message in one protocol might be equivalent, in terms of Bytes, to many smaller-sized packets of another protocol.

An additional parameter for evaluating the efficiency of a Top-k query processing algorithm is with regards to the *Completion Time*, which is defined as the interval between the receipt of the query's end-of-transmission message by a target node, and the query response's end-of-transmission message re-

Table 1
The local scores of five objects $o_1..o_5$ which are located at nodes $v_1..v_5$. The last column displays the overall rank which is computed as the sum of scores.

v_1	v_2	v_3	v_4	v_5	TOP-5
oid,val	oid,val	oid,val	oid,val	oid,val	oid,val
o_3, .99	o_1, .91	o_1, .92	o_3, .74	o_3, .67	o_3, 4.05
o_1, .66	o_3, .90	o_3, .75	o_1, .56	o_4, .67	o_1, 3.63
o_0, .63	o_0, .61	o_4, .70	o_2, .56	o_1, .58	o_4, 2.07
o_2, .48	o_4, .07	o_2, .16	o_0, .28	o_2, .54	o_0, 1.88
o_4, .44	o_2, .01	o_0, .01	o_4, .19	o_0, .35	o_2, 1.75

ceived by the host node that issued the query. This parameter is particularly important in online systems that need to have access to the query answer in an online manner.

Note that the above parameters are not sufficient to quantify the intrinsic characteristics of distributed environments. In cases where the network suffers from failures, it is also required to study the *Average Error*, that is, the error of the acquired answer-set from the correct results. Studying such a parameter is important as many messages might fail to be delivered to the querying node, thus leading to an erroneous answer. The objective of a distributed Top-k algorithm is to keep the error parameter as small as possible.

Section 5, will present an assessment of these parameters on the algorithms we describe in this paper.

2.3 Outline of Operation

We shall next outline the motivation behind the three phases of the Threshold Join Algorithm described in this paper. For ease of exposition, we will use an example network of five nodes ($v_1..v_5$), where each node maintains locally the scores of five objects ($o_0..o_4$). The score values for each node are presented on Table 1. We shall also utilize an example query that seeks to identify the object (i.e., K=1) that maximizes the sum of local scores across all nodes (in the given example that would be o_3 with score $4, 05$).

A *Querying Node (QN)*, also referred to as *Sink* in this work, can easily compute an answer to the top-1 query by first transferring all o_{ij} pairs to itself, then perform a local join, and finally extracting the desired results. The message complexity of this approach is clearly $\Theta(m \cdot n)$. Assuming that each node transmits its local scores to *QN* either directly or over a multi-hop network, this approach would require $\sum_{i=0}^{n} \delta(v_i) * m$ messages, where $\delta(v_i)$ is the depth of node v_i from *QN*. Obviously this algorithm, denoted as the *Centralized Join*

Algorithm (CJA), is extremely expensive in practice (this will be validated in experimental section 5).

However when intermediate nodes perform some local aggregation similarly to [26], then this cost can be reduced to $\Theta(n)$ which is essentially one message per edge (let this be denoted as the *Staged Join Algorithm (SJA)*). Note that in this scenario, each node packs in a single message exactly m tuples (values) similarly to the TAG [26] in-network aggregation mechanism deployed in the TinyDB [27] database system for Wireless Sensor Networks. An important problem is that in our environment the query window value m might be arbitrary large, which therefore constitutes this approach as an expensive alternative.

The Threshold Join Algorithm studied in this paper is an efficient Top-k query processing algorithm for distributed environments. TJA decreases the number of objects that are required to be transmitted from each node, by using a probing and filtering phase before proceeding with the retrieval of the final answer-set. Recall that each node v_i essentially maintains a list of scores which is denoted as $list(v_i)$. In addition, the algorithm seeks to optimize the use of the network resources by pushing computation into the network.

More specifically, the algorithm consists of three phases:

(1) The *Lower Bound* phase, in which the querying node finds a lower bound on the lists by probing the nodes in the network;
(2) The *Hierarchical Joining* phase, in which each node uses the lower bound for eliminating the objects that are below this bound and join the qualifying objects with results coming from children nodes; and
(3) The *Clean-Up* phase, which is invoked if k objects could not be computed exactly in the previous phases.

2.4 Our Contribution

In this paper we consider an in-network processing technique for efficient Top-k query evaluation in distributed computing networks. The key idea of our work is to transmit only the necessary objects towards the querying node and to perform the score calculation procedure in the network rather than in a centralized way. This imposes many problems and challenges, as the attributes are located on different nodes in the network, which are not directly accessible. More specifically:

- We study the feasibility and applicability of Top-k queries in distributed settings;

- We propose the TJA algorithm, which is a novel algorithm to resolve Top-k queries in a hierarchical environment using a fixed number of phases. Our algorithm deploys in-network query processing in order to reduce network traffic, conserve energy and increase application performance.
- We perform a large-scale evaluation using a real testbed of 75 workstations along with a variety of real and realistic datasets. For this purpose we have implemented a middleware system over which we evaluate various algorithms and design choices. Our results demonstrate the efficiency of the TJA algorithm under a diverse number of settings.

3 The Threshold Join Algorithm (TJA)

In this section we will describe in detail TJA, an efficient algorithm that utilizes a non-uniform threshold on the queried attribute in order to minimize the transfer of data when a query is executed. Additionally, *TJA* resolves queries in the network rather than in a centralized fashion which further minimizes the consumption of bandwidth and delay. It is important to notice that the TJA algorithm is not a technique for performing general-purpose database joins in a distributed setting but it rather only utilizes this operator in its execution.

Without loss of generality, we assume that the query is disseminated to the nodes in the network prior to the execution of the algorithm. For ease of exposition, we shall supplement the description with an example execution that utilizes the values of Table 1.

3.1 Description of Algorithm

The three phases of the algorithm are presented below:

Lower Bound (LB) Phase

This phase identifies a set of objects that are used to construct a threshold. The Top-k results are guaranteed to have a score above this threshold.

The phase works by having each node v_i sort in descending similarity order the elements in its local list (i.e., $list(v_i)$). v_i then extracts from this list the identifiers of the k local highest-ranked objects (we denote this new set as $list_k(v_i)$). If v_i is a leaf node, then it transmits $list_k(v_i)$ to its *parent* node. A parent node is a node which provides a path to the querying node *QN*. If v_i is a non-leaf node, then it waits until it receives $list_k(v_j)$ from all of its children

v_j, at which point it performs a union of all collected lists. This procedure creates the following partial lower bound $L(v_i)$:

$$L(v_i) = list_k(v_i) \bigcup (\bigcup_{\forall j \in children(v_i)} list_k(v_j))$$

$L(v_i)$ is then propagated to the parent of v_i and when this recursive operation terminates, the querying node will receive a list of object identifiers that define the complete lower bound $L_{queryNode}$=L_{total}=$\{l_1, l_2, \ldots, l_o\}$, $o \geq k$.

Consider our working example of Table 1 (which is for convenience also summarized in Figure 3d). Assume that the querying node is v_0, and that it issues the top-1 query: *"Find the time moment with the highest average temperature."*. We note that it is easy to express such a query in our framework, by setting $sim(q_j, o_{ij})$ equal to the normalized temperature of node i at time j (this is equivalent to asking for the time moment with a vector closest to (1,..,1)).

In Figure 3a, we illustrate the Query Spanning Tree (QST) of the LB phase. Each node sends its top value: $list_1(v_i)$ for the vertices v_1, v_2, v_3, v_4, v_5 is o_3, o_1, o_1, o_3 and o_3 respectively. Consequently, the lower bound in our working example is L_{total}=$\{o_1, o_3\}$. The figure shows how each intermediate node v_i calculates the partial lower bound $L(v_i)$. For example at node v_4, we perform the following union $L(v_4) = \{o_3 \cup o_3\}$, therefore v_4 is only required to propagate o_3 to v_2 rather than all the pairs in its local list.

Hierarchical Join (HJ) Phase

In the second phase the querying node propagates L_{total} to all nodes in the network. Each node then uses L_{total} for eliminating the objects that are below this bound.

Specifically, the sink starts out by disseminating L_{total} to all nodes in the network using the same hierarchy created in the LB phase. This has a message complexity of $\Theta(n)$, where n is the number of nodes in the network. Each node receiving L_{total}, searches its local sorted $list(v_i)$ in order to identify the index of the lowest ranked object that belongs to L_{total}.

More precisely, a procedure *FindMinRank* locates the lowest ranked object that belongs to L_{total}. All objects above idx are candidates for the result.

In the next step, each node uses the locally generated idx in order to extract all tuples above idx from its sorted $list(v_i)$. Let $list_{idx}(v_i)$ denote the set of o_{ij} pairs generated by this procedure. If a node is a leaf node, it simply forwards $list_{idx}(v_i)$ towards its parent. Otherwise, a node waits until it receives all $list_{idx}(v_j)$ from all of its children v_j, at which point it performs a full outer

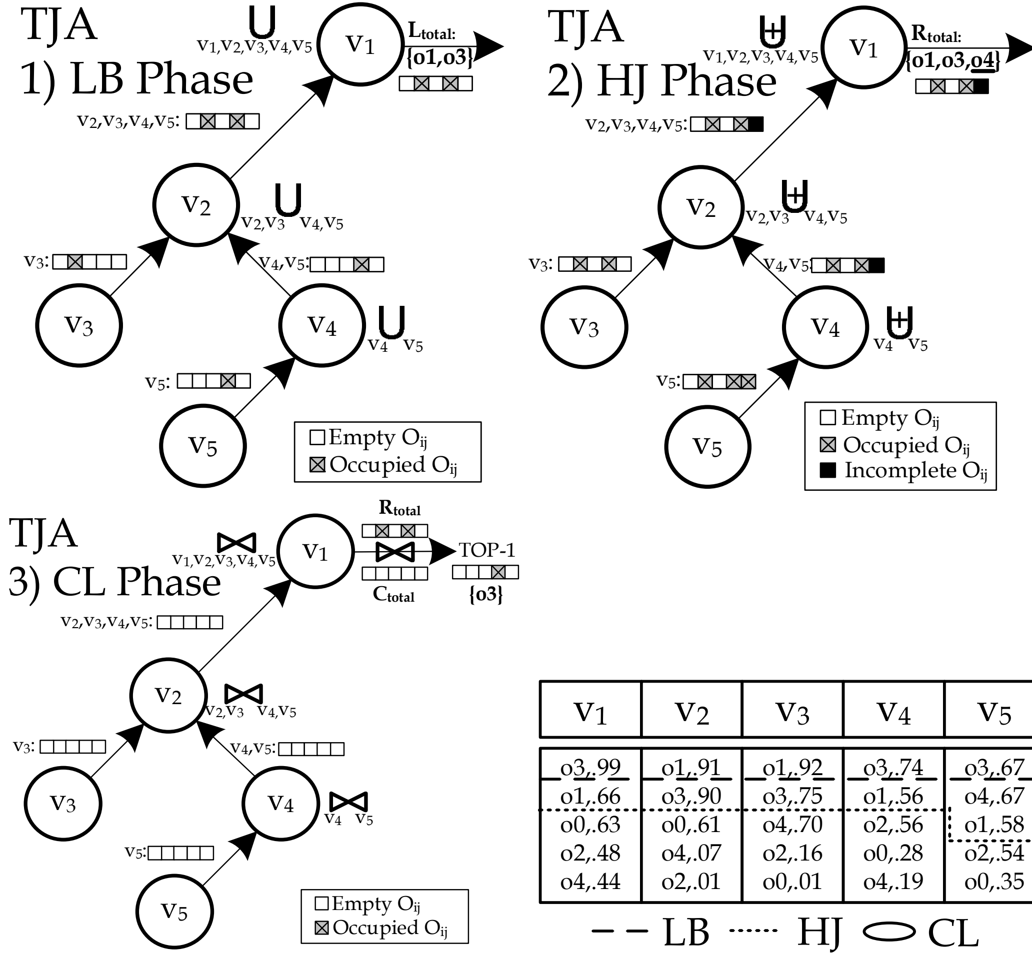

Fig. 3. The Query Spanning Tree for the three phases of the TJA Algorithm: 1) Lower Bound (LB), 2) Hierarchical Join (HJ) and 3) Clean-Up (CL) Phase. The table shows the objects qualifying in each phase.

1: **procedure** FINDMINRANK($L_{total}, list(v_i)$)
2: $idx = -1$
3: **for** $j = 1$ to $|L_{total}|$ **do**
4: $pos = rank(L_{total}[j], list(v_i))$
5: **if** $(idx < pos)$ **then**
6: $idx = pos$
7: **end if**
8: **end for**
9: return idx
10: **end procedure**

join using the *FullOuterJoin* procedure illustrated next. We note that in a full outer join of two relations A and B, in addition to the rows that join on the objectID, the rows of both A and B without a match also appear in the result.

However, the rows that do not match in both A and B, are marked with a *incomplete* flag. Below we present how *FullOuterJoin* ($\biguplus$) works:

$$R(v_i) = list_{idx}(v_i) \biguplus (\biguplus_{\forall j \in children(v_i)} list_{idx}(v_j))$$

The above procedure creates a local *partial result* $R(v_i)$. During this computation the algorithm computes a partial score for each object o_j in $R(v_i)$. If object o_j appears in the result list of v_i and in the result list of all its children this partial score can be computed exactly using the above function. If on the other hand, o_j does not appear in all of the lists (and so it is marked as *incomplete* by the *FullOuterJoin* procedure), the algorithm computes an upper bound estimation on the score.

In particular, if object o_l does not appear in the result list of some node v_i, we compute an upper bound on the partial score of o_l by substituting $sim(q_i, o_{il})$ with $min_{a \in list_{idx}(v_i)}[sim(q_i, a)]$. Note that $min_{a \in list_{idx}(v_i)}[sim(q_i, a)]$ represents the maximum possible value that $sim(q_i, o_{il})$ might have. In other words, all values in the table $list(v_i)$ below $min_{a \in list_{idx}(v_i)}[\ sim(q_i, a)]$ are smaller than $sim(q_i, o_{il}))$. This is attributed to the fact that $list(v_i)$ is sorted in descending similarity order.

To understand the above description consider node v_4 in Figure 3 (HJ Phase). Also assume that node v_5 has just completed the transmission of its three objects to v_4 (i.e., o_3, o_4 and o_1). Both o_3 and o_1 are directly joined with the values of v_4 yielding thus an exact score for each object. On the other hand, object o_4 does now not appear in $list_{idx}(v_4)$. Consequently, v_4 substitutes o_4 with the maximum possible value that this object might get in $list(v_4)$, which is equal to .56 (i.e., the value of o_1).

Before forwarding the complete and incomplete result list, v_i marks those score computations that are upper bounds using an extra bit to differentiate them from the exact partial scores. v_i then forwards $R(v_i)$ to its parent and when this recursive operation terminates, the querying node will receive a superset of the final Top-k result $R_{queryNode}$=R_{total}=$\{r_1, r_2, \ldots, r_o\}$, $o \geq k$.

In our working example the superset result is $R_{total} = \{(o_1, 3.63)$, $(o_3, 4.05)$, $(o'_4, 3.54)\}$ and the querying node just extracts the highest rank answer o_3 which is the result with the highest score. In Figure 3b, we illustrate the QST for the HJ phase of our working example. The figure shows how each intermediate node v_i calculates the partial result $R(v_i)$. At node v_4, the full outer join of $list_{idx}(v_4) = \{(o_3, .74)$, $(o_1, .56)\}$ and $list_{idx}(v_5) = \{(o_3, .67)$, $(o_4, .67)$, $(o_1, .58)\}$ generates the following partial result $R(v_4) = \{(o_3, 1.41)$, $(o_1, 1.14)$, $(o'_4, 1.23)\}$, where the result of o_4 is marked as an upper bound while the others are exact results.

Clean-Up (CL) Phase

In the last phase of the algorithm the querying node has collected a list of objects for which either the *complete score* or an upper bound of its score has been computed (*incomplete score*). If k objects have been found with an exact score above the largest incomplete object, then the algorithm can safely terminate and return those objects as the answer.

On the other hand, if less than k exact scores were computed or if some of the incomplete scores is higher than the exact scores, then QN has to request the exact scores for the remaining incomplete objects that are above the kth exact score. This is achieved by sending a clean-up message to all nodes in the system. The clean-up message contains the list of objects incompletely computed (denoted as R'). Each node receiving the request uses the function $objects_{R'}(v_i)$, in order to locate the objects in R' which were not shipped to the sink. Each node waits, as before, for the clean-up results that might arrive from its children nodes, before proceeding to the full score calculation. This is achieved by using the below function:

$$C(v_i) = objects_{R'}(v_i) \bowtie (\bowtie_{\forall j \in children(v_i)} objects_{R'}(v_j))$$

When the querying node receives C_{total}, it computes the final top-k answers. This is achieved by joining C_{total} with R_{total}, where R_{total} is the partial result generated in the HJ phase.

In our example the querying node has calculated an upper bound of 3.54 for o_4, which is less than the score of o_3 (i.e., 4.05), and so the querying node does not have to execute the CL phase. In the experimental results of Section 5, we can observe that the TJA algorithm completes in two phases for the most of the cases and that the CL phase is rarely necessary.

3.2 TJA Correctness

We shall now prove the correctness of the TJA algorithm and show that it generates the desired answer in every instance. In particular, we need to prove that the TJA algorithm does not miss any object in the final top-k results irregardless of the input values.

Before proceeding to the correctness proof of TJA, let us show that the Upper-Bound Mechanism, utilized in the second (HJ) phase of the algorithm, is correct.

Upper-Bound Mechanism: *This mechanism is invoked in the second (HJ) phase of the TJA algorithm, in response to a Top-k query* $Q = \{q_1, \ldots, q_n\}$*, and works as follows: For any object* o_l *that does not appear in the* $list_{idx} v_i$ *of some node* v_i *substitute* $sim(q_i, o_{il})$ *with* $min_{a \in list_{idx}(v_i)}[sim(q_i, a)]$*. Clearly,* $min_{a \in list_{idx}(v_i)}[sim(q_i, a)]$ *is always larger or equal than* $sim(q_i, o_{il})$*, as* $list(v_i)$ *(*$\forall v_i \in V$*) is sorted in descending similarity order and because object* o_l *is below position idx in* $list(v_i)$ *(*$\forall v_i \in V$*).*

Lemma 1: *Invoking the Upper-Bound Mechanism in the TJA algorithm yields a correct Top-k answer-set.*

Proof: To prove this lemma, let us consider the final answer-set returned in response to Q, denoted as *Result*. Our analysis shows that *Result* comprises of the following two cases:

(1) **Case 1:** *Result* contains $k+\delta$ $(0 \leq \delta \leq m-k)$ objects with "exact" scores (let this subset be defined as K) and every object in K has a larger score than the rest $|Result - K|$ objects with "incomplete" (bounded above) scores.

For this case, the Upper-Bound mechanism is correct as every "incomplete" object o'_i in the $(Result - K)$ set, has already its maximum possible score and the score o'_i is by definition of this case smaller than the scores of objects in K. Consequently, the TJA algorithm can correctly terminate in the second (HJ) phase without missing any correct results, as the score of every o'_i will be smaller than those objects returned as an answer to Q.

(2) **Case 2:** *Result* contains $k - \delta$ $(0 < \delta \leq k)$ objects with "exact" scores (let this subset be defined again as K) and the rest $|Result - K|$ objects with "incomplete" (bounded above) scores might have any value (i.e., larger, smaller or even equal to the values in K).

For this case, the Upper-Bound mechanism is again correct as the score of every "incomplete" object o'_i in the $(Result - K)$ set will be determined exactly by the third (CL) phase of the TJA algorithm. Consequently, the final step of the algorithm is to identify the k highest scores in a list of $|Result|$ exact scores which is straightforward. □

We shall next prove that the TJA algorithm, which utilizes the Upper Bound mechanism, is correct in every instance.

Lemma 2: *The TJA algorithm returns the k answers with the highest score to a given query Q in its three phases.*

Proof (direct): Let $Q = \{q_1, \ldots, q_n\}$ denote a Top-k query resolved with the TJA algorithm, r_1 an arbitrary object not returned in response to Q (i.e.,

$r_1 \in Results$) and r_2 an arbitrary object that is not returned in response to Q (i.e., $r_2 \notin Result$). We want to show that $Score(r_2) < Score(r_1)$ always holds.

First note that $Score(r_1)$ is defined exactly either during the second (HJ) phase or the third (CL) phase of the TJA algorithm using the upper-bound mechanism that was proven before. In any case, the minimum possible value of $Score(r_1)$ is equal to the summation of minimum values in the individual $list_{idx}$ sets of each node (formally, that would be equivalent to $\sum_{i=1}^{n} min_{a \in list_{idx}(v_i)}$ $[sim(q_i, a)]$, denoted as τ for convenience). On the other hand, the maximum possible value of $Score(r_2)$ is at most $\tau - \epsilon$, where $\epsilon \in \mathbb{R}$ and $\epsilon > 0$, since $r_2 \notin Results$ (this would be the first element below the minimum value in the local $list_{idx}$ of each node). Otherwise, r_2 would have been included in the HJ phase of the TJA execution (and cleared in the CL phase if necessary). If that was the case though, this object would have been handled by the Upper Bound mechanism. Consequently, it holds that $Score(r_2) < \tau$ and it directly follows that $Score(r_2) < Score(r_1)$ as $Score(r_1) \geq \tau$ $\square$

4 Experimental Evaluation Methodology

In this Section we describe our experimental framework and the datasets that we utilized in our evaluation. We focus on the *efficiency*, that is, the number of transmitted *Bytes*, the number of *Messages* and required *Time* to obtain the final result. In all cases, the presented algorithms return exactly the same result with what is returned by evaluating the query over a centralized collection of lists. In cases where the network suffers from failures, we also study the *Average Error* of the algorithms, that is, the error of the acquired answer-set from the correct results. Detailed definitions of these parameters were given in Section 2.2.

4.1 Description of Algorithms

In order to make a meaningful assessment of the TJA algorithm, we have implemented and tested a number of algorithms with a real middleware system that we developed in JAVA and deployed over a network of 75 workstations.

We chose as a baseline the results we obtained by three other algorithms that operate over exact scores, produce exact results and require a fixed number of phases. In particular, we have implement: i) The *Centralized Join Algorithm (CJA)*, where each node transmits its local scores to the querying node *QN*; ii) The *Staged Join Algorithm (SJA)*, where each intermediate node performs local aggregation on the intermediate results in order to minimize messag-

ing and response time and iii) The *Three-Phase Uniform Threshold (TPUT)*, proposed by Cao and Wang in [8], that was the first fixed-round algorithm designed for single-hop networks.

TPUT uses, similarly to our approach, a three phase protocol in order to resolve Top-k queries in distributed networks. The algorithm starts out by constructing a uniform bound across all lists. In our working example of Table 1, this bound would be constructed by fetching the first row from each node. The Querying Node (QN) then constructs the partial aggregate scores for the two objects seen so far: $o_1 = .91 + .92 = 1.83$ and $o_3 = .99 + .67 + .74 = 2.4$. These scores are *partial*, because they do not take into account the score from every node in the system. In the second phase, QN uses these partial scores to define a uniform threshold as $\tau = ((\text{Kth lowest partial score})/n)$, where n is the number of nodes in the system. This score, which in our working example is equal to $\tau = 240/5 = .48$, is then disseminated to all nodes in the network. Each node then transmits to QN all objects with scores above τ, in particular, v_1, v_2, v_3, v_4 and v_5 transmit every object with score above $o_2 = .48, o_0 = .61, o_4 = .70, o_2 = .56$ and $o_2 = .54$ respectively (i.e., an upward projection). The final phase of the algorithm is only invoked if k objects could not be computed exactly.

The disadvantage of TPUT over our TJA algorithm, is that the threshold τ is uniform for all nodes and this results in excessive messaging as we will see in the experiments that follow this next.

4.2 Experimental Infrastructure

This section describes our middleware system that allows us to study Top-k and other data retrieval algorithms in large-scale P2P systems. Our system runs on a network of 75 workstations (each hosting a number of nodes). Each workstation has an AMD Athlon 800MHz-1.4GHz processor with memories varying from 256MB-1GB RAM, running Mandrake Linux 8.0 (kernel 2.4.3-20) all interconnected with a 100MB LAN. Our middleware is written entirely in Java and comes along with an extensive set of UNIX shell scripts that allow the easy deployment and administration of the system.

Our middleware consists of three components: (i) *graphGen* which generates random network topologies and configuration files for the various nodes participating in a given experiment, (ii) *topkPeer* which is a P2P client that implements a binary data protocol of the algorithms we discuss in this paper, and (iii) *searchPeer* which is a P2P client that performs queries and harvests answers back from a deployed network. We use random graphs with uniform degree distributions instead of power-law topologies because in our setting

nodes have equal capabilities. A network of 1000 nodes can be launched in approximately 20 seconds.

Data Transfer Protocol: We have implemented a binary data protocol, as opposed to using a text-based protocol, which makes the transmission of messages and data lists highly efficient. In our configuration each message has a 23 bytes header, which includes the unique identifier of the message, the payload size (i.e., how many objects are packed in the message), as well as other implementation specific parameters. Each element in $list(v_i)$ (i.e., an o_{ij} pair), requires 8 byte (oid=4 bytes, val=4 bytes) and these pairs are consecutively packed after the message header. Each *topkPeer* v_i maintains open TCP socket connections with its neighbors so there is no need to establish a new connection every time v_i wants to send a message to one of its neighbors. This saves our nodes from the three-way TCP handshake and other overheads associated with the connection establishment procedure. Notice that by keeping the TCP sockets open in our setting is just an additional improvement to our middleware system and not a prerequisite for the correctness of our algorithm. In fact, all implemented Top-k algorithms would equally benefit from this improvement.

Finally, in our setting several peer nodes were running on the same workstation and all processes were assigned low priorities because we used instructional lab workstations. Therefore the presented time results are pessimistic bounds on what can be achieved in a dedicated environment.

4.3 Description of Datasets

Our experiments are based on the following two datasets:

1) NetMon - Server Infrastructure: This is a synthetic dataset that simulates the 1GB main memory system of 1000 workstations on which users schedule various processes. In this dataset the local memory utilization at consecutive time moments features a similar behavior. We did not measure the actual memory utilization of the machine on which each node was running, because in our setting several nodes were running on the same workstation. Therefore the memory utilization of each machine was always at near maximum utilization.

Additionally, we assume that the workstations are located at different geographic locations, similarly to the PlanetLab testbed [9], and that monitoring the workstations using a centralized setting becomes very expensive. Therefore we use a Peer-to-Peer architecture in which nodes can communicate only with

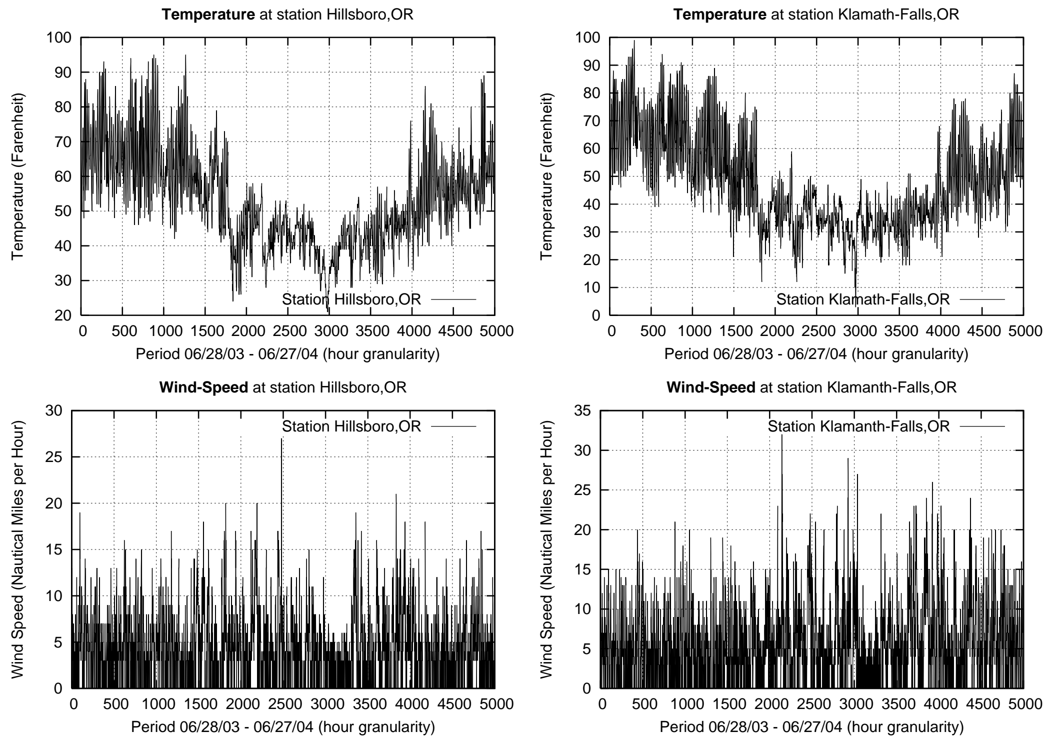

Fig. 4. Temperature (top) and Wind-Speed (Bottom) timeseries for two data logging stations in Port Angeles (WA) and Hillsboro (OR).

a few other close-by nodes (e.g., using their longitude/latitude coordinates).

2) AtmoMon - Atmospheric Data: This is a real dataset of atmospheric data collected at 32 sites in the Washington and Oregon states, by the Department of Atmospheric Sciences at the University of Washington. [1] More specifically, each of the 32 sites maintains the average temperature and wind-speed on an hourly basis for 208 days between June 2003 and June 2004 (i.e., 4990 time moments). Figure 4 shows two timeseries from two different logging stations. The figures indicate that in the temperature timeseries, consecutive time moments are correlated (feature a self-similar pattern). However, the wind-speed timeseries fluctuates and that distorts the performance benefits of Top-k query processing algorithm as we will show in the experimental section.

5 Experimental Evaluation

In this section we describe a series of experiments to investigate the efficiency of the TJA algorithm compared to CJA, SJA and TPUT. In each subsection

[1] http://www-k12.atmos.washington.edu/k12/grayskies/

Table 2
Configuration parameters for the instances of our experimental evaluation.

Section	Objective	Dataset	m	k	n	d
5.1	TJA vs. (CJA, SJA)	NetMon	2,5K-10K	3	1,000	10
5.2	TJA vs. (CJA, SJA)	NetMon	10,000	1-300	1,000	10
5.3	TJA vs. (CJA, SJA)	NetMon	10,000	3	1,000	4-10
5.4	TJA vs. (CJA, SJA)	AtmoMon	4,990	3	1,000	4
5.5	TJA vs. (CJA, SJA)	AtmoMon	4,990	3	32	4
5.6.1	TJA vs. TPUT	NetMon	25,000	5	25-100	5
5.6.2	TJA vs. TPUT	AtmoMon	4,990	3	32	4

we study the impact of different values for the variables m, k and d, which are the parameters for the query window size, the number of expected results and bushiness of network topology respectively. The first class of experiments aims to present the advantages of a three-phase algorithm (i.e., TJA) compared to the other one-phase algorithms (i.e., CJA and SJA). The second class of experiments only focuses on comparing the three-phase algorithms (i.e., TJA vs. TPUT).

For the first class of experiments we present the following: In subsections 5.1-5.3 we present the first series of experiments which is based on the *NetMon* dataset. The query is to find the k moments in the last m minutes, in which the n servers (each with a degree d) had the highest average memory utilization in total. In subsection 5.4-5.5, we present the second experimental series which evaluates the same parameters on the *AtmoMon* dataset. The query for this series was to find the three moments on which the average *temperature* among all monitors was maximum. We also study the impact of failures on the performance of the algorithm we propose. For the second class of experiments we present in subsection 5.6 the third experimental series which compares the TJA algorithm to the TPUT algorithm using the NetMon and AtmoMon datasets.

Table 2 summarizes the configuration parameters for the subsequent sections. In our experiments, all measurements are averages over 10 consecutive runs.

5.1 Performance Comparison of One-Phase vs. Three-Phase Algorithms

In the first experimental series we investigate the performance of the TJA algorithm compared to the other one phase algorithms discussed in this paper (i.e., CJA and SJA). We define performance using three criterions: 1) *Bytes* used in order to send the results, 2) *Time* waited before the Top-k results become available at the querying node and 3) *Messages* consumed. Our query

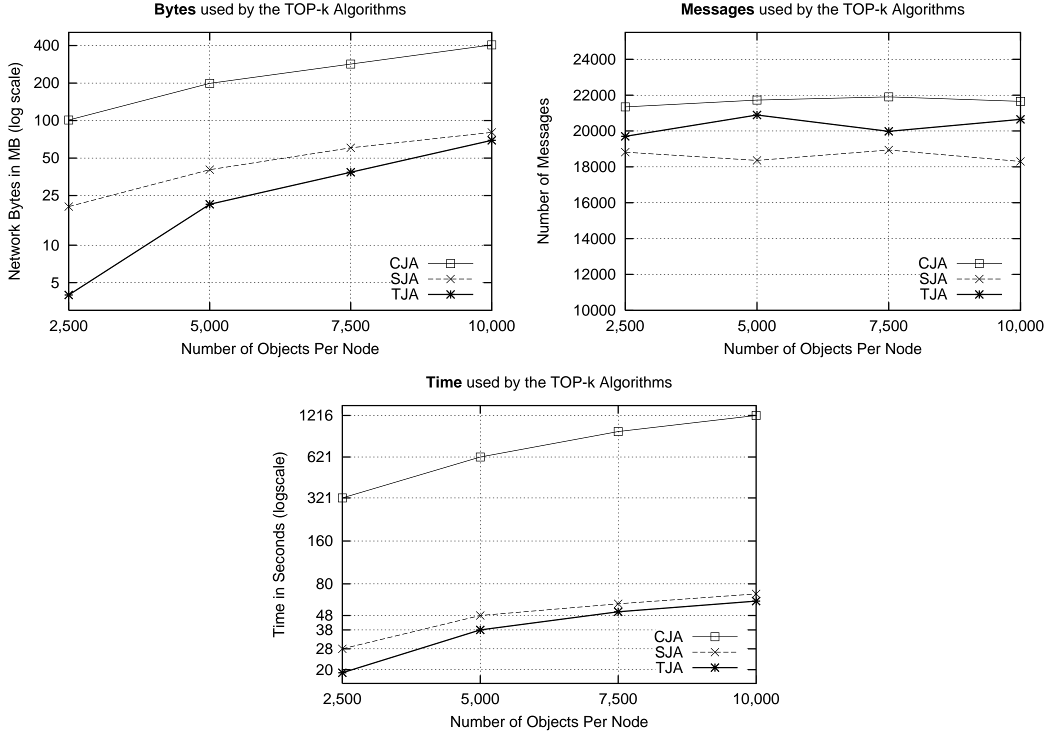

Fig. 5. **Performance Evaluation:** Comparing TJA vs. two one-phase counterparts (CJA and SJA) using 1000 NetMon nodes with average degree d=10. The graphs illustrate the: a) Bytes consumed (top,left), b) Messages transmitted (top,right) and c) Query completion Time (bottom). The figures show that the number of bytes and the total time to receive back the answers is significantly decreased with TJA.

in the first experiment is to find the top-3 moments in the last 2.5K, 5K, 7.5K and 10K moments at which the infrastructure had the highest average memory utilization in total.

Bytes: In Figure 5 (top,left), we plot the number of bytes required for the different windows of time. As we can see with 2.5K objects, CJA consumes two orders of magnitudes more network bytes than the TJA algorithm and one order of magnitude more bytes for the rest cases. Another drawback of CJA is that even after the querying node receives all o_{ij} pairs, it requires several seconds before it can join the 1000 lists and calculate the final result. Finally, the figure also indicates that TJA always outperforms the SJA algorithm. Note that when the parameter m increases, the TJA algorithm gets much coarser bounds in the LB phase and that decreases the performance gap between TJA and SJA. Yet, the performance of the TJA algorithm is still superior than both SJA and CJA for any m size in the range 2,500 to 10,000.

Messages: In Figure 5 (top,right), we plot the total number of messages, including the initial brute-force search, required by each algorithm. Note that

a message in our setting has a variable size [2]. Therefore the total number of messages presented in the case of TJA, is a mixture of smaller and larger sized messages. The graph indicates that TJA requires approximately 2000 messages more than SJA. This was expected as in the absence of the CL phase, TJA requires only one additional round-trip which costs only $2.(n-1)$ messages. However since each peer maintains open socket connections with its neighbors, the additional round-trips can be conducted by circumventing the three-way TCP handshake and the other overheads associated with the connection establishment procedure. Our results show that although TJA uses many smaller size messages the overall number of bytes is significantly decreased. Next we present the advantage of the TJA algorithm with regards to the overall querying time.

Time: In Figure 5 (bottom), we plot the time required by the different algorithms. As we can see TJA always outperforms the other two one-phase alternatives, even though it uses three phases. This happens because TJA's HJ phase is much smaller than the monolithic and time consuming one phase presented in the rest algorithms. Note that TJA's other two phases (LB and CL) are in practice very short. For example for 5K objects, the LB phase required 10 sec; the HJ phase 22 sec and the CL phase 0 sec. Therefore this was much lower than the 48 seconds required by the SJA algorithm and 621 seconds required by CJA.

5.2 Scalability with Respect to k

In the second experiment, we evaluated the efficiency of the TJA Algorithm with respect to the parameter k. More specifically, we increase the parameter k while keeping the number of objects constant at m=$10,000$. We omit the time and messages graphs since both parameters used 60ms and 20K messages respectively for all runs.

Figure 6 (left) indicates that TJA works well even when the parameter k reaches the 3% of the number of objects. More specifically, with $k = 300$ TJA requires 79MB, which is still better than what SJA requires (i.e., 80.4MB). When the parameter k reaches the 5% of the number of objects at each node, SJA outperforms TJA as it again requires 80.4MB while TJA requires 84MB for its three phases. Therefore when the parameter k is very large it might be cheaper to gather all results at the querying node rather than using TJA. Finally note that CJA and SJA require a constant amount of bytes for any parameter of k as both algorithms transfer a priori all lists to the querying node.

[2] A message has a 23 bytes header and up-to 2^{32} o_{ij} pairs (8 bytes each).

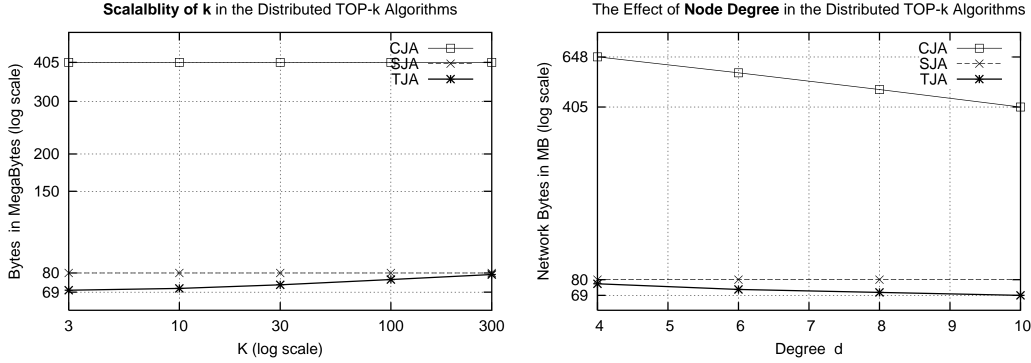

Fig. 6. **Evaluating Scalability and Network Topology:** a) Scalability with respect to the parameter k, b) Increasing the Query Spanning Tree hierarchy.

5.3 Increasing the Query Tree Hierarchy

In the third experiment we vary the average degree d of the nodes while keeping constant the rest parameters. Since the number of nodes in the network remains constant and the degree increases, the network becomes more bushy (densely connected) and the diameter of the graph decreases. More specifically the diameter has the values 9, 6, 6 and 5 for degrees 4, 6, 8 and 10 respectively.

Figure 6 (right) indicates that CJA requires as much as 648MB when $d = 4$ while SJA and TJA require 80.4MB and 77MB respectively. This was expected as the cost in CJA is a function of the depth of the tree which is significantly larger for smaller degree parameters. The figure also indicates that TJA is similarly favored by larger d parameters as more aggregation is achieved at intermediate nodes. For example when $d = 4$ the cost is 77MB, while with $d = 10$ the cost reduces to 69MB. Finally, since SJA requires to transmit for any spanning tree $m * n$ o_{ij} pairs its performance is independent of d.

5.4 Experimentation with Real Data

In this section we present our results using real data from the *AtmoMon* dataset (presented in section 4.3). Our query is to find the three hours (moments), on which the average *temperature* among all monitors was maximum. We set the degree parameter to $d = 4$ in order to create an adequately deep (but connected) network of diameter=6.

In Figure 7 (top,left with failure threshold f=0) [3], we plot the number of required bytes for the three algorithms when there are no failures in the network. The figure indicates that TJA requires an order of magnitude less bytes

[3] The rest f values will be discussed in the next subsection.

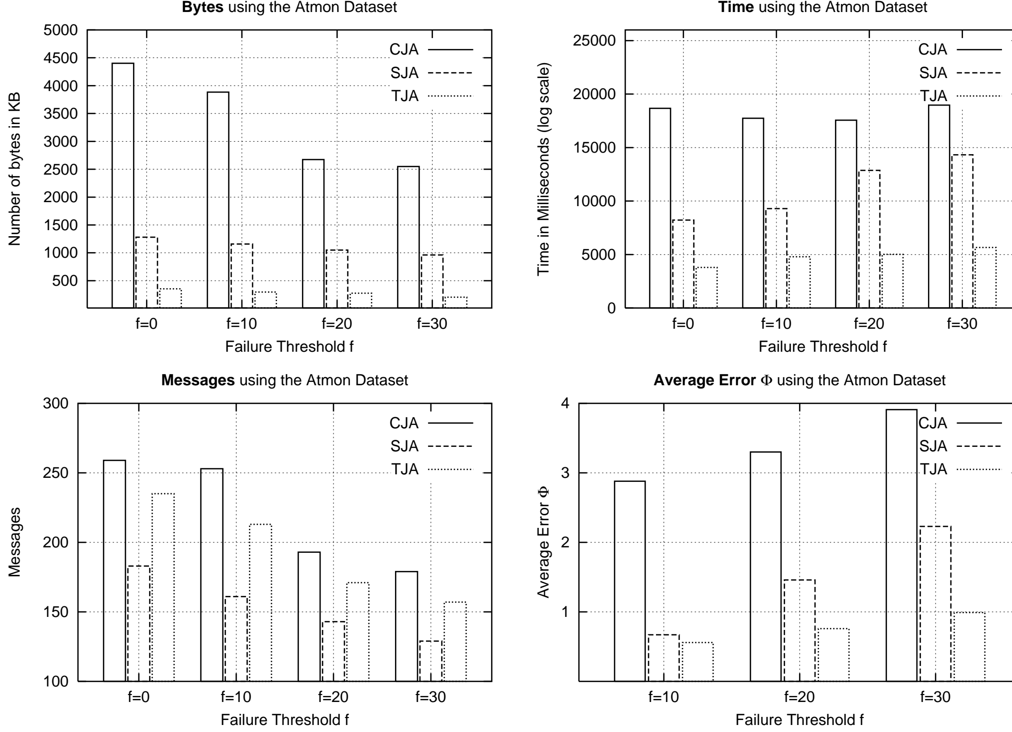

Fig. 7. **Performance Evaluation under Failures:** a) Bytes and b) Time, c) Messages and d) Average Error Φ for failure various thresholds (0.0, 0.1, 0.2 and 0.3).

than SJA for calculating the results. Figure 7 (top,right with f=0), displays the respective time for the three algorithms. As we can see TJA calculates the Top-k result in only 3,797 ms (LB=1,059ms, HJ=2,730ms and CL=8ms) while SJA requires 8,224 ms and CJA 18,666 ms. Finally in Figure 7 (bottom,left with f=0), we plot the number of messages. The figure indicates again that TJA requires more messages than SJA. More specifically TJA requires 246 messages while CJA requires 259 messages and SJA 183 messages. However it is again important to mention that most of the messages used by TJA are small in size.

We have also experimented with the AtmoMon wind-speed dataset. Our results reveal that TJA has not a big advantage over its competitor SJA. More specifically, TJA requires 1.23MB while SJA requires 1.27MB before the top 3 results become available. After analyzing the timeseries of various nodes we observed that the wind-speed values do not have a correlated behavior between consecutive moments. For example in Figure 4, we plotted the values generated at two atmospheric data logging stations in Port Angeles, WA, USA and Hillsboro, OR, USA. We observe that the temperature presents some correlated behavior between consecutive moments while the wind-speed fluctuates. The uncorrelated behavior of the wind-speed, makes the TJA algorithm to obtain a much coarser lower bound in the LB-phase which results in the transmission

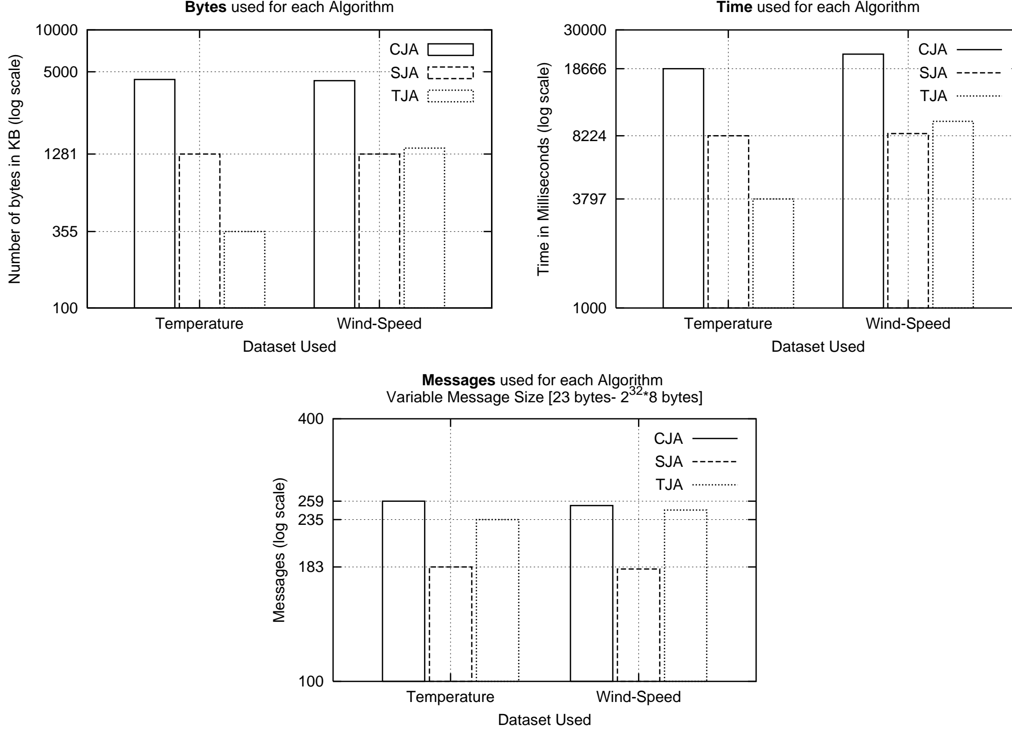

Fig. 8. **Performance Evaluation using Real Datasets:** We utilize the temperature and wind-speed timeseries gathered by 32 weather monitoring stations in Washington and Oregon, USA.

of large lists during the HJ phase.

5.5 Experimentation under Failures

Node failures and abnormal disconnections generate a dynamic environment in which nodes are leaving and joining the network in an ad-hoc manner. This does not allow the querying node to obtain the correct Top-k results. Therefore we define the *Average Error* function Φ, that measures the error in the rank of the obtained results compared to the real Top-k results.

Formally Φ is defined as:

$$\Phi = \frac{1}{k} * \sum_{i=0}^{k} penalty(o_i) \tag{2}$$

where $penalty(o_i)$ is equal to $realrank(o_i) - rank(o_i)$ if $realrank(o_i) > rank(o_i)$ or zero otherwise (higher *score* denotes higher *rank*). Note that $realrank(o_i)$ is obtained by running a Top-k algorithm locally while $rank(o_i)$ is obtained from the results coming from the network. In order to simulate failures in our

stable environment, we randomly and uniformly disconnect nodes from the network and join them back after some fixed interval. At any given moment the percentage of disconnected nodes is no more than a given threshold f.

For this experimental series we utilize an *AtmoMon* network and show the number of bytes, the required time and the number of messages utilized by each of the described techniques when the failure factor is set to 10%, 20% and 30% respectively. The plots in Figure 7 indicate that in all techniques the number of bytes and messages decreases linearly with increasing failures because more nodes tend to loose their parents.

On the other hand the required time to execute a query under failures increases for SJA and TJA. This happens because each node v_i has a timer $\tau_{v_i}=\tau_{max} * ttl(v_i)$, where τ_{max} is a given threshold and $ttl(v_i)$ the time-to-live parameter taken from the query when it arrives at v_i. This decreases the waiting period of nodes deeper in the QST hierarchy. Nodes in CJA on the other hand do not use any timer as messages might arrive from an arbitrary large number of nodes (not just the children) and therefore the waiting time remains fixed.

In 7 (bottom,right) we plot the Φ parameter for the different failure thresholds. The figure shows that TJA always achieves excellent resilience $\Phi < 1$. This means that the rank of each result is on average less than one position wrong. On the other hand CJA and SJA have a larger Φ value because they take longer to complete and therefore miss more results. Our results suggest that TJA is tolerant to failures which might an important attribute in networks with high error rates (such as sensor networks).

5.6 Performance Comparison of Three-Phase Algorithms: TJA vs. TPUT

In the final experiment we study the performance of the TJA algorithm against the TPUT algorithm presented in Section 4.1. Recall that the TPUT algorithm uses, similarly to our approach, a three phase protocol in order to resolve Top-k queries in distributed environments. The TPUT algorithm starts out by constructing a uniform bound across all lists. It then computes a uniform threshold τ and fetches all objects above this bound. The disadvantage of TPUT over TJA is that the threshold τ is uniform for all nodes rather than non-uniform. This results in the unnecessary transfer of many objects that are not in the final top-k result.

We will present two different experiments for this series: i) comparison under the NetMon dataset and ii) comparison under the AtmoMon dataset.

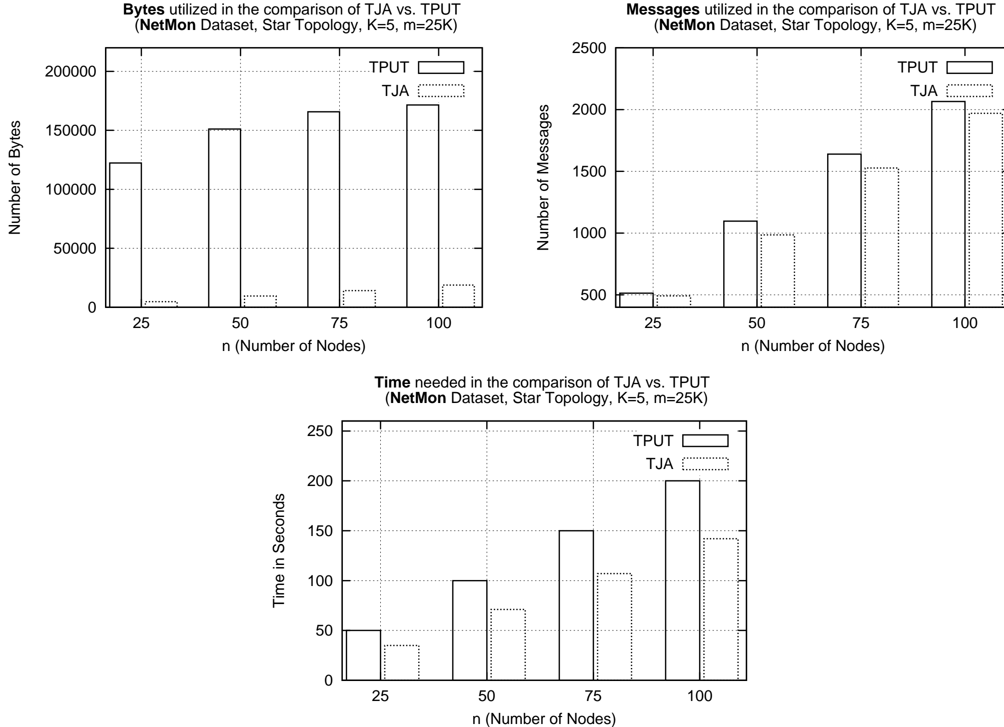

Fig. 9. **Performance** comparison of TJA vs. TPUT using the **NetMon** dataset

5.6.1 NetMon Dataset

This subsection presents the performance characteristics of the TJA and TPUT algorithms when these algorithms are executed over a NetMon dataset that consists of 25,000 objects. Our query is to find the top-5 answers for varying number of nodes n {25,50,75 and 100}.

Bytes: Figure 9 (top-left), demonstrates that the TJA algorithm has an order of magnitude better performance than TPUT (i.e., 152,709 Bytes vs. 11,755 Bytes). By carefully examining the execution we observe that this major performance gap is attributed to deficiency of the uniform threshold established in the second phase of the TPUT algorithm. In particular, in TPUT each node goes much further down its own local score list than the TJA algorithm.

Messages: In Figure 9 (top-right), we present the number of application-layer messages that these two algorithms require. The figure shows that the performance of the given algorithms is very close in regards to this parameter. Note that both of these algorithms are 3-phase algorithms, consequently, both of them require approximately the same number of application-layer messages to complete the query. On the contrary, the number of messages at the lower layers of the communication stack (e.g., TCP/IP packets), are much larger for TPUT and that has already been justified in the Bytes graphs in Figure 9 (top-left). We speculate that the slight performance difference of the given algorithms with respect to the messages parameter is attributed to random

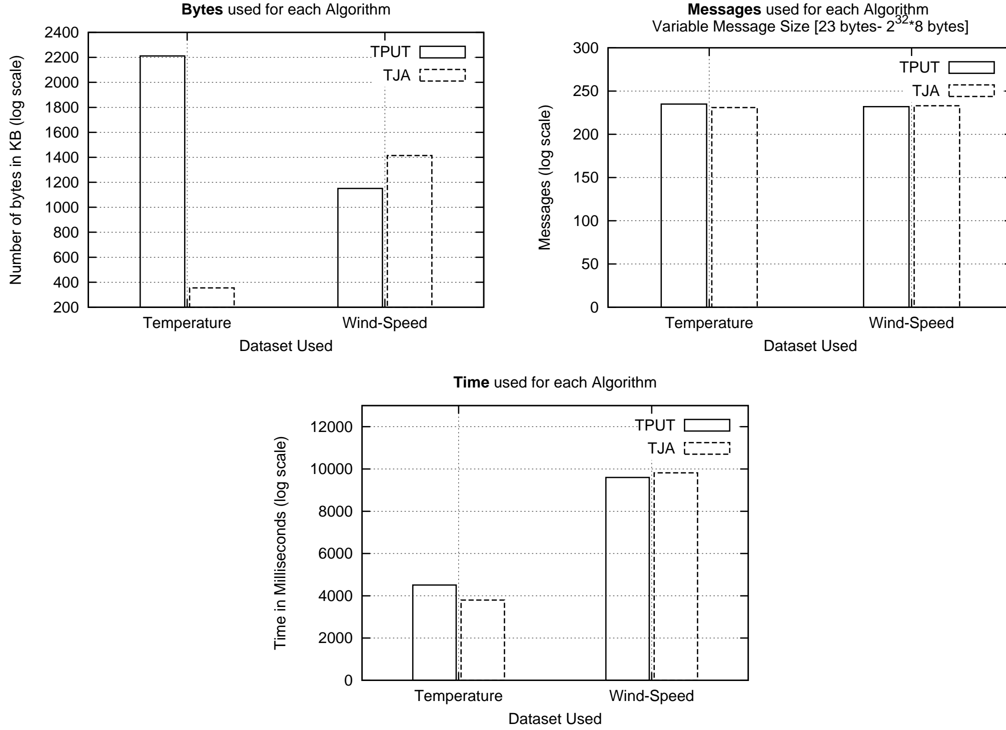

Fig. 10. **Performance** comparison of TJA vs. TPUT using the **AtmoMon** dataset

failures and not to a specific advantage of the TJA algorithm over TPUT.

Time: Finally in Figure 9 (bottom), we present the amount of time required by the two algorithms. The figure again shows that the TJA algorithm has a significant advantage over TPUT with regards to this parameter (i.e., on average 89 sec. vs. 125 sec). This is again attributed to the fact that the TJA algorithm transmits considerably less bytes than the TPUT algorithm during its second phase. Consequently, in TJA the user will be able to retrieve the answer back much faster than in TPUT (e.g., for n=100 the difference is as big as 58 seconds).

5.6.2 AtmoMon Dataset

This subsection presents the performance characteristics of the TJA and TPUT algorithms when these are executed over the AtmoMon temperature dataset and the AtmoMon wind-speed dataset. Both of these datasets consist of m=4990 objects distributed over n=32 nodes. Our query is to find the top-3 answers.

Bytes: Figure 10 (top-left), demonstrates that the TJA algorithm has again an order of magnitude better performance than TPUT for the temperature dataset (i.e., TPUT requires 2,212KB while TJA only 355KB). On the other hand, the performance under the wind-speed dataset is reversed (i.e., TPUT consumes 1,151KB while TJA consumes 1,414KB). As mentioned earlier, the

temperature dataset presents some correlated behavior between consecutive moments while the wind-speed dataset fluctuates. The uncorrelated behavior of the wind-speed, provides an advantage to the TPUT algorithm which proceeds with the retrieval of the largest part of each node's local list in its second phase. On the other hand, the TJA algorithm obtains a much coarser lower bound in LB-phase which results in the retrieval of many additional values during the HJ and CL phases.

Messages: In Figure 10 (top-right), we present the number of application-layer messages that these two algorithms require. The figure shows that the performance of the given algorithms is again very close in regards to this parameter. Yet, the TJA algorithm proceeds with less objects than TPUT.

In particular, for the temperature dataset the TPUT algorithm requires 235 messages while the TJA algorithm requires only 231 messages. For the wind-speed dataset the situation is reversed, with TPUT requiring 232 messages and TJA 233 messages. We again speculate that the difference in the messages parameter is attributed to random failures.

Time: Finally in Figure 10 (bottom), we present the amount of time required by the two algorithms. The figure again shows that the TJA algorithm is significantly faster for the temperature dataset (where significantly more objects are transferred to the sink node) and comparable for the wind-speed dataset. In particular, for the temperature dataset we obtained the following measurements on average (for our 10 repetitions): 4,507ms for TPUT and 3,797ms for TJA. For the wind-speed dataset we obtained the following measurements on average: 9,598ms for TPUT and 9,815ms for TJA.

6 Related Work

There has been a lot of work in the area of Top-k query processing in the database community [13,15,28,4,5,7,12,19,22,16,3,17,30,31,25,35,36]. Most of these solutions have been developed for either centralized database management scenarios or client-server settings. An excellent survey of these algorithms appears in [20]. In this section we will focus our attention on exact answers that are generated by sporadic distributed Top-k queries which are executed over a multi-hop communication network. Top-k query processing over a multi-hop network is of particular importance in this work, firstly because the multi-hop network is the fundamental assumption underlying the operation of many distributed environments such as sensor and p2p networks.

The *Fagin Algorithm (FA) [13]*, is one of the first Top-k query processing algorithms over middleware systems which interact with a number of autonomous

data sources. In *FA*, a querying node *QN* performs a two phase retrieval which consists of a sorted access and random access phase. Initially, *QN* accesses the n lists in parallel until it locates k objects which belong to all lists. In our working example, this would be equivalent to retrieving the first two rows of the table (as $k = 1$ and o_3 now belongs to all lists). In the random phase, *QN* requests from each node to send the score for any object whose score could not be computed exactly (i.e., o_1 and o_4). Assuming a star topology, FA has with very high probability a cost of $O(m^{(n-1)/n} \cdot k^{1/n})$. A main problem with FA, is that the initial sorted phase accesses all lists at the same depth. This results in the retrieval of a large number of unnecessary object scores. Additionally, FA cannot be executed efficiently in a multi-hop environment, because the sorted phase is iterative. As a result, each iteration translates into a large number of communication messages.

The most widely recognized algorithm for Top-k queries in a centralized environment is the *Threshold Algorithm (TA)* [13]. *TA* starts out by performing a parallel sorted access to the n lists. While an object o_i is seen by *QN*, *TA* performs a random access to the other lists to find the exact score for o_i (i.e., $\sum_{j=1}^{n} o_{ij}$). In our working example it would first resolve exactly the two objects in the first row (i.e., $o_3 = 4.05$ and $o_1 = 3.63$). After finding the exact score for each object in the current row [4], it computes a *threshold* value τ as the sum of all exact scores in the current row (i.e., $\tau = .99 + .91 + .92 + .74 + .67 = 4.23$). Since τ is larger than both scores of o_3 and o_1, the TA algorithm performs another iteration in which the threshold τ is refined as the sum of scores across the second line. The algorithm stops after k objects have been found with a score above τ. While the *TA* algorithm accesses less objects than *FA*, it also uses more round trips as it invokes several small random accesses. This would again translate into an arbitrary large number of phases, which is highly undesirable for a distributed environment.

The advantages of our TJA algorithm over other query processing algorithms, such as TA and FA, are twofold:

(1) Instead of performing random accesses for individual objects, TJA performs them all together in the clean-up phase. This minimizes the number of messages, and therefore also the number of transmitted bytes [5]. Additionally, it also minimizes the delay to find the expected answer.
(2) By structuring the communication into three phases, it increases in-network aggregation. This happens because individual random accesses yield less aggregation than the aggregation achieved by a few combined random accesses.

The *TPUT (Three-Phase Uniform Threshold)* algorithm, proposed by Cao and

[4] Sorted access is executed on a row-at-a-time basis
[5] Note that each message has a fixed overhead (a header)

Wang in [8], was the first fixed-round algorithm designed for single-hop networks. TPUT has been overviewed in Section 4.1 of this paper and Section 5, has presented an experimental assessment of its performance. The disadvantage of TPUT, over our TJA algorithm, is that the threshold τ is uniform for all nodes. This results in the transfer of an unnecessarily large number of objects that are not in the final result. On the contrary, the TJA algorithm uses an non-uniform threshold which is different for each node, and which provides the algorithm with a higher pruning capability. Note that the TPUT algorithm utilizes a value as a threshold, while we utilize a set of objects. As a result, our solution can prune away non-qualifying objects more efficiently. One final point is that the original TPUT idea only focus on single-hop topologies rather than on multi-hop topologies, as we do and consequently does not exploit any further in-network optimizations such as the upper-bound mechanism we proposed.

Yu et. al., presented in [37] three fixed round algorithms coined *TPAT (Three-Phase Adaptive-Threshold)*, *TPOR (Three-Phase Object-Ranking)*, and *HT (Hybrid-Threshold)*. The first algorithm TPAT, is identical with TPUT in the first and the last phase, while in its second phase, it computes a non-uniform threshold, rather than a uniform one, as follows: In the second phase *QN* uses the partial scores from the first phase to divide the threshold τ into $\tau_1, \tau_2, ..., \tau_n$ based on summary statistics sent from nodes in the network. These statistics are represented as equi-depth histograms. The collection of τ thresholds is then disseminated back to the nodes on the network, similarly to TPUT. In that respect, the TPAT algorithm is a direct improvement of the TPUT algorithm, that fixes the uniform threshold problem.

The TPOR [37] algorithm is similar to the TJA algorithm, in a sense that both algorithms disseminate a set list that defines a threshold rather than a uniform threshold, but the TPOR algorithm does not deploy the hierarchical upper-bound mechanism devised by the TJA algorithm.

The third algorithm, HT, combines the advantages of TPOR and TPAT in the following manner: During the second phase, the central manager *QN*, requests from all the nodes to send any object with a score higher than a hybrid threshold, calculated as the maximum of the uniform threshold obtained from TPAT and the threshold obtained from TPOR. Although this heuristic has been shown to perform well in many occasions, it introduces an additional round trip, differentiating it therefore from all the aforementioned algorithms which were three-phase, rather than four-phase, algorithms.

Top-k algorithms have also been studied in other settings, where the pruning of the retrieval space is highly desirable (Table 3 provides a taxonomy). Bruno et al. [7] discuss the problem of answering Top-k queries over web accessible databases. Babcock and Olston [4] consider the problem of continually pro-

Table 3
A taxonomy of popular distributed Top-k query processing algorithms.

Algorithm	**Phases**	**Scores**	**Result**	**Query**
FA/TA[13]	Undefined	Exact	Exact	On-Demand
TPUT[8], TPAT [37]	3	Exact	Exact	On-Demand
TJA [40], TPOR [37]	3	Exact	Exact	On-Demand
HA[37]	4	Exact	Exact	On-Demand
BABOLS[4]	Contin.	Approx	Approx	Continuous
MINT[38]	Contin.	Approx	Exact	Continuous
PROB[35]	Iterative	Approx	Approx	On-Demand
UB-K/UBLB-K[42]	Iterative	Approx.	Exact	On-Demand
KLEE[30]	3-4	Approx	Approx	On-Demand

viding approximate top-k answers using a technique we denote as *BABOLS*. The problem is tackled by installing arithmetic constraints at each node which define the current Top-k scores at any point. However the results are approximate and continuous while our solution is designated for on-demand queries that request the exact result. The continuous case was later also extended to a hierarchical sensor network environment [11] and [29]. A similar approach also appears in [38] but the given framework focuses on exact answers in wireless sensor networks. Note that monitoring of threshold functions and frequent items over data streams has recently been an intense area of research [24,23,32].

The authors in [30], examine the problem of approximate Top-k queries in distributed environments. The paper assumes that each node maintains an approximation of the local scores instead of the actual scores. The approximation essentially consists of an equi-width histogram on the local scores along with a bloom filter [6] per histogram bucket which captures object identifiers inserted into the specific bucket. Therefore, their framework is appropriate for approximate and on-demand queries over approximate scores while the UB-K/UBLB-K [42] algorithms address a similar case but with exact rather than approximate answers. A sampling-based approach to optimize Top-k queries in sensor networks is also the core topic in [33].

In [5,21], the problem of identifying the Top-k objects from relations which are horizontally fragmented over peers in a P2P environment is studied. The proposed solution depends on each peer having knowledge of the total score of each object that it manipulates. This is not possible for vertically partitioned relations, as this requires access to all relations in their entirety, which constitutes their approach inapplicable in our context.

Finally, most of the above approaches assume a star (or single-hop) communication topology, in which all nodes are directly accessible by the querying

entity. On the other hand, our work has focused on the challenges of a hierarchical (or multi-hop) topology.

7 Conclusion & Future Work

In this paper we have studied Top-k query processing algorithms for distributed environments. The objective of a Top-k query is twofold: To find the k highest-ranked answers to a user defined similarity function, and to minimize some cost metric associated with the retrieval of the correct answers. *TJA* uses a non-uniform threshold on the queried attribute in order to minimize the number of tuples that have to be transferred towards the querying node so that the correct answer can be computed. Additionally, *TJA* resolves queries in the network rather than in a centralized fashion, which further minimizes the consumption of bandwidth and delay.

Our extensive experimental evaluation using our middleware testbed reveals that the algorithm we propose is both efficient and practical. Specifically, our results indicate that *TJA* consumes an order of magnitude less network bytes than other alternatives, scales well with respect to the parameter k and the network topology. Our study also reveals that the algorithm executes faster than its competitors and works well in dynamic environments.

We believe that our algorithm will be a useful component for query processing engines of distributed data management systems in the future. In the future we plan to extend the prototype of our algorithm that has been implemented in the context of sensor network context [2]. We additionally plan to make our middleware infrastructure publicly available.

References

[1] Akamai Inc., Available at: http://www.akamai.com/

[2] Andreou P., Zeinalipour-Yazti D., Vassiliadou M., Chrysanthis P.K., Samaras G., "KSpot: Effectively Monitoring the K Most Important Events in a Wireless Sensor Network", In *Proceedings of the 25th International Conference on Data Engineering (ICDE)* (demo), Shanghai, China, May 29 - April 4, 2009.

[3] Amato G., Rabitti F., Savino P. and Zezula P., "Region proximity in metric spaces and its use for approximate similarity search", In *ACM Transactions on Information Systems*, Vol. 21, Iss. 2, pp.: 192-227, 2003.

[4] Babcock B. and Olston C., "Distributed Top-K Monitoring", In *Proceedings of the ACM SIGMOD international conference on Management of data*

(SIGMOD), San Diego, CA, USA, Pages 28-39, 2003.

[5] Balke W.-T., Nejdl W., Siberski W., Thaden U., “Progressive Distributed Top-K Retrieval in Peer-to-Peer Networks”, In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pp.: 174-185, Tokyo, Japan, 2005.

[6] Bloom B. H., “Space/Time Trade-Offs in Hash Coding with Allowable Errors”, Communication of the ACM, 13(7):422-426, 1970.

[7] Bruno N., Gravano L. and Marian A., “Evaluating Top-K Queries Over Web Accessible Databases”, In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, San Jose, CA, USA, Page 369, 2002.

[8] Cao P. and Wang Z., “Efficient Top-K Query Calculation in Distributed Networks”, In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing (PODC)*, St. John’s, Newfoundland, Canada, Pages 206-215, 2004.

[9] Chun B.N., Culler D.E, Roscoe T., Bavier A.C, Peterson L.L, Wawrzoniak M., Bowman M., “PlanetLab: an overlay testbed for broad-coverage services”, Computer Communication Review Volume 33, Issue 3, Pages 3-12, 2003.

[10] Claffy K., Tracie E., McRobb D. “Internet tomography”, 1999.

[11] Deligiannakis A., Kotidis Y., Roussopoulos N., “Hierarchical in-Network Data Aggregation with Quality Guarantees”, In *9th International Conference on Extending Database Technology (EDBT)*, Heraklion, Greece, Pages 658-675, 2004.

[12] Donjerkovic D. and Ramakrishnan R., “Probabilistic Optimization of Top-N Queries”, In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pp. 411-422, Edinburgh, Scotland, UK, 1999.

[13] Fagin R., “Combining Fuzzy Information from Multiple Systems”, In *Proceedings of the fifteenth ACM symposium on Principles of database systems (PODS)*, Montreal, Quebec, Canada, Pages 216-226, 1996.

[14] Fagin R., “Fuzzy Queries In Multimedia Database Systems”, In *Proceedings of the seventeenth ACM symposium on Principles of database systems (PODS)*, Seattle, WA, USA, pp. 1-10, 1998.

[15] Fagin R., Lotem A. and Naor M., “Optimal Aggregation Algorithms For Middleware”, In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, Santa Barbara, CA, USA, Pages 102-113, 2001.

[16] Gravano L. and Chaudhuri S., “Evaluating Top-K Selection Queries”, In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, UK, Pages 397-410, 1999.

[17] Guntzer U. , Balke W., Kiebling W., “Optimizing Multi-Feature Queries for Image Databases”, In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, pp. 419 - 428, 2000.

[18] Hansen, T., Otero, J., McGregor, A., Braun, H-W., “Active measurement data analysis techniques”, In *CIC*, Las Vegas, Nevada, 2000.

[19] Ilyas I.F., Aref W.G. and Elmagarmid A.K., “Supporting Top-k Join Queries in Relational Databases”, In *the International Journal on Very Large Data Bases (VLDBJ)*, Vol. 13 , Iss. 3, pp. 207-221, 2003.

[20] Ilyas I.F., Beskales G., and Soliman M.A., “A Survey of Top-k Query Processing Techniques in Relational Database Systems”, In *ACM Computing Surveys*, Volume 40, Issue 4, Article 11, October 2008.

[21] Kalnis P., Ng W-S., Ooi B-C., Tan K-L., “Answering similarity queries in peer-to-peer networks”, In *Proceedings of the 13th international World Wide Web (WWW)*, Pages 482-483, New York City, NY, USA, 2004.

[22] Li C., Chang K., Ilyas I., Song S., “RankSQL: Query Algebra and Optimization for Relational Top-k Queries”, In *Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD)*, Baltimore, Maryland, 2005.

[23] Olston C., Jiang J., Widom J., “Adaptive Filters for Continuous Queries over Distributed Data Streams”, In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD)*, San Diego, California, Pages: 563-574, 2003.

[24] Manjhi A., Shkapenyuk V., Dhamdhere K., Olston C., “Finding (Recently) Frequent Items in Distributed Data Streams”, In *“Proceedings of the 21st International Conference on Data Engineering (ICDE*, Tokyo, Japan, Pages: 767-778, 2005.

[25] Nepal S., Ramakrishna M. V., “Query Processing Is-sues in Image (Multimedia) Databases”, In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, Page: 22, Washington, DC, USA, 1999.

[26] Madden S.R., Franklin M.J., Hellerstein J.M., Hong W., “TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks”, In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, Boston, MA, pp. 131-146, 2002.

[27] Madden S.R., Franklin M.J., Hellerstein J.M., Hong W., “The Design of an Acquisitional Query Processor for Sensor Networks”, In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD)*, San Diego, CA, USA, Pages 491-502, 2003.

[28] Marian A., Gravano L., Bruno N., “Evaluating Top-k Queries over Web-Accessible Databases”, In *ACM Transactions on Database Systems (TODS)*, Vol 29, Num. 2, pp. 319-362, June 2004.

[29] Mouratidis K., Bakiras S., Papadias D., “Continuous monitoring of top-k queries over sliding windows”, In *Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD)*, Chicago, IL, USA Pages: 635-646, 2006.

[30] Michel S., Triantafillou P., Weikum G., "KLEE: A Framework for Distributed Top-K Query Algorithms", In *31st conference in the series of the Very Large Data Bases (VLDB)*, pp. 637-648, 2005, Trondheim, Norway, 2005.

[31] Nejdl W., Siberski W., Thaden U. and Balke W., "Top-k Query Evaluation for Schema-Based Peer-to-Peer Networks", In *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, Hiroshima, Japan, November 2004.

[32] Sharfman I., Schuster A., Keren D., "A geometric approach to monitoring threshold functions over distributed data streams", In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD)*, Pages: 301-312, Chicago, IL, USA, 2006.

[33] Silberstein A., Braynard R., Ellis C., Munagala K., Yang J., "A Sampling-Based Approach to Optimizing Top-K Queries in Sensor Networks", In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, Washington, DC, USA, Page: 68, 2006.

[34] Szewczyk R., Osterweil E., Polastre J., Hamilton M., Mainwaring A.M., Estrin D., "Habitat monitoring with sensor networks", Commun. ACM 47(6): 34-40 (2004).

[35] Theobald M., Schenkel R., Weikum G., "Top-k Query Evaluation with Probabilistic Guarantees", In *Proceedings of the Thirtieth international conference on Very large data bases (VLDB)*, Toronto, Canada, pp.: 648-659, 2004.

[36] Xiong L., Chitti S., Liu L., "Top-k Queries across Multiple Private Databases", In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp: 145-154, 2005.

[37] Yu H., Li H., Wu P., Agrawal D., Abbadi A.E., "Efficient Processing of Distributed Top-k Queries", In *Proceedings of the 16th International Conference on Database and Expert Systems (DEXA)*, Krakow, Poland, pp. 65-74, 2005.

[38] Zeinalipour-Yazti D., Andreou P., Chrysanthis P., Samaras G., "MINT Views: Materialized In Network Top-k Views in Sensor Networks", In *Proceedings of the 8th International Conference on Mobile Data Management (MDM)*, pp. 182-189, Mannheim, Germany, 2007.

[39] Zeinalipour-Yazti D., Neema S., Gunopulos D., Kalogeraki V. and Najjar W., "Data Acquision in Sensor Networks with Large Memories", In *Proceedings of the Intl. Workshop on Networking Meets Databases (NetDB)*, collocated with ICDE'05, Tokyo, Japan, 2005.

[40] Zeinalipour-Yazti D., Vagena Z., Gunopulos D., Kalogeraki V., Tsotras V., Vlachos M., Koudas N., Srivastava D., "The Threshold Join Algorithm for Top-K Queries in Distributed Sensor Networks", In *Proceedings of the 2nd international workshop on Data management for sensor networks (DMSN)*, collocated with VLDB'05, Trondheim, Norway, Vol. 96, Pages: 61-66, 2005.

[41] Zeinalipour-Yazti D., Lin S., Kalogeraki V., Gunopulos D., Najjar W., "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices", In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, pp. 31-44, 2005.

[42] Zeinalipour-Yazti D., Lin S., Gunopulos D., "Distributed Spatio-Temporal Similarity Search", In *Proceedings of the ACM 15th Conference on Information and Knowledge Management (CIKM)*, Arlington, VA, USA, pp. 14-23, 2006.