MINERVA∞: A Scalable Efficient Peer-to-Peer Search Engine

Sebastian Michel[1], Peter Triantafillou[2], and Gerhard Weikum[1]

[1] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
{smichel, weikum}@mpi-inf.mpg.de
[2] R.A. Computer Technology Institute and University of Patras, 26500 Greece
peter@ceid.upatras.gr

Abstract. The promises inherent in users coming together to form data sharing network communities, bring to the foreground new problems formulated over such dynamic, ever growing, computing, storage, and networking infrastructures. A key open challenge is to harness these highly distributed resources toward the development of an ultra scalable, efficient search engine. From a technical viewpoint, any acceptable solution must fully exploit all available resources dictating the removal of any centralized points of control, which can also readily lead to performance bottlenecks and reliability/availability problems. Equally importantly, however, a highly distributed solution can also facilitate pluralism in informing users about internet content, which is crucial in order to preclude the formation of information-resource monopolies and the biased visibility of content from economically-powerful sources. To meet these challenges, the work described here puts forward MINERVA∞, a novel search engine architecture, designed for scalability and efficiency. MINERVA∞ encompasses a suite of novel algorithms, including algorithms for creating data networks of interest, placing data on network nodes, load balancing, top-k algorithms for retrieving data at query time, and replication algorithms for expediting top-k query processing. We have implemented the proposed architecture and we report on our extensive experiments with real-world, web-crawled, and synthetic data and queries, showcasing the scalability and efficiency traits of MINERVA∞.

1 Introduction

The peer-to-peer (P2P) approach facilitates the sharing of huge amounts of data in a distributed and self-organizing way. These characteristics offer enormous potential benefit for the development of internet-scale search engines, powerful in terms of scalability, efficiency, and resilience to failures and dynamics. Additionally, such a search engine can potentially benefit from the intellectual input (e.g., bookmarks, query logs, click streams, etc.) of a large user community participating in the sharing network. Finally, but perhaps even more importantly, a P2P web search engine can also facilitate pluralism in informing users about internet content, which is crucial in order to preclude the formation of information-resource monopolies and the biased visibility of content from economically powerful sources.

G. Alonso (Ed.): Middleware 2005, LNCS 3790, pp. 60–81, 2005.

Our challenge therefore was to exploit P2P technology's powerful tools for efficient, reliable, large-scale content sharing and delivery to build a P2P web search engine. We wish to leverage DHT technology and build highly distributed algorithms and data infrastructures that can render P2P web searching feasible.

The crucial challenge in developing successful P2P Web search engines is based on reconciling the following high-level, conflicting goals: on the one hand, to respond to user search queries with high quality results with respect to precision/recall, by employing an efficient distributed top-k query algorithm, and, on the other hand, to provide an infrastructure ensuring scalability and efficiency in the presence of a very large peer population and the very large amounts of data that must be communicated in order to meet the first goal.

Achieving ultra scalability is based on precluding the formation of central points of control during the processing of search queries. This dictates a solution that is highly distributed in both the data and computational dimensions. Such a solution leads to facilitating a large number of nodes pulling together their computational (storage, processing, and communication) resources, in essence increasing the total resources available for processing queries. At the same time, great care must be exercised in order to ensure efficiency of operation; that is, ensure that engaging greater numbers of peers does not lead to unnecessary high costs in terms of query response times, bandwidth requirements, and local peer work.

With this work, we put forward MINERVA∞, a P2P web search engine architecture, detailing its key design features, algorithms, and implementation. MINERVA∞ features offer an infrastructure capable of attaining our scalability and efficiency goals. We report on a detailed experimental performance study of our implemented engine using real-world, web-crawled data collections and queries, which showcases our engine's efficiency and scalability. To the authors' knowledge, this is the first work that offers a highly distributed (in both the data dimension and the computational dimension), scalable and efficient solution toward the development of internet-scale search engines.

2 Related Work

Recent research on structured P2P systems, such as Chord [17], CAN [13], Skip-Nets [9] or Pastry [15] is typically based on various forms of distributed hash tables (DHTs) and supports mappings from keys to locations in a decentralized manner such that routing scales well with the number of peers in the system. The original architectures of DHT-based P2P networks are typically limited to exact-match queries on keys. More recently, the data management community has focused on extending such architectures to support more complex queries [10,8,7]. All this related work, however, is insufficient for text queries that consist of a variable number of keywords, and it is absolutely inappropriate for full-fledged Web search where keyword queries should return a ranked result list of the most relevant approximate matches [3].

Within the field of P2P Web search, the following work is highly related to our efforts. Galanx [21] is a P2P search engine implemented using the Apache HTTP server and BerkeleyDB. The Web site servers are the peers of this architecture; pages are stored only where they originate from. In contrast, our approach leaves it to the peers to what extent they want to crawl interesting fractions of the Web and build their own local indexes, and defines appropriate networks, structures, and algorithms for scalably and efficiently sharing this information.

PlanetP [4] is a pub/sub service for P2P communities, supporting content ranking search. PlanetP distinguishes local indexes and a global index to describe all peers and their shared information. The global index is replicated using a gossiping algorithm. This system, however, appears to be limited to a relatively small number of peers (e.g., a few thousand).

Odissea [18] assumes a two-layered search engine architecture with a global index structure distributed over the nodes in the system. A single node holds the complete, Web-scale, index for a given text term (i.e., keyword or word stem). Query execution uses a distributed version of Fagin's threshold algorithm [5]. The system appears to create scalability and performance bottlenecks at the single-node where index lists are stored. Further, the presented query execution method seems limited to queries with at most two keywords. The paper actually advocates using a limited number of nodes, in the spirit of a server farm.

The system outlined in [14] uses a fully distributed inverted text index, in which every participant is responsible for a specific subset of terms and manages the respective index structures. Particular emphasis is put on minimizing the bandwidth used during multi-keyword searches. [11] considers content-based retrieval in hybrid P2P networks where a peer can either be a simple node or a directory node. Directory nodes serve as super-peers, which may possibly limit the scalability and self-organization of the overall system. The peer selection for forwarding queries is based on the Kullback-Leibler divergence between peer-specific statistical models of term distributions.

Complementary, recent research has also focused into distributed top-k query algorithms [2,12] (and others mentioned in these papers which are straightforward distributed versions/extensions of traditional centralized top-k algorithms, such as NRA [6]). Distributed top-k query algorithms are an important component of our P2P web search engine. All these algorithms are concerned with the efficiency of top-k query processing in environments where the index lists for terms are distributed over a number of nodes, with index lists for each term being stored in a single node, and are based on a per-query coordinator which collects progressively data from the index lists. The existence of a single node storing a complete index list for a term undoubtedly creates scalability and efficiency bottlenecks, as our experiments have showed. The relevant algorithms of MINERVA∞ ensure high degrees of distribution for index lists' data and distributed processing, avoiding central bottlenecks and boosting scalability.

3 The Model

In general, we envision a widely distributed system, comprised of great numbers of peers, forming a collection with great aggregate computing, communication, and storage capabilities. Our challenge is to fully exploit these resources in order to develop an ultra scalable, efficient, internet-content search engine.

We expect that nodes will be conducting independent web crawls, discovering *documents* and computing *scores* of documents, with each score reflecting a document's importance with respect to *terms* of interest. The result of such activities is the formation of *index lists*, one for each term, containing relevant documents and their score for a term. More formally, our network consists of a set of nodes N, collectively storing a set D of documents, with each document having a unique identifier $docID$, drawn from a sufficiently large name space (e.g., 160 bits long). Set T refers to the set of terms. The notation $|S|$ denotes the cardinality of set S. The basic data items in our model are triplets of the form *(term, docID, score)*. In general, nodes employ some function $score(d, t) : D \rightarrow (0, 1]$, which for some term t, produces the score for document d. Typically, such a scoring function utilizes tdf*idf style statistical metadata.

The model is based on two fundamental operations. The $Post(t, d, s)$ operation, with $t \in T$, $d \in D$, and $s \in (0, 1]$, is responsible for identifying a network node and store there the (t, d, s) triplet. The operation $Query(T_i, k) : return(L_k)$, with $T_i \subseteq T$, k an integer, and $L_k = \{(d, TotalScore(d)) : d \in D,\ TotalScore(d) \geq RankKscore\}$, is a top-k query operation. $TotalScore(d)$ denotes the aggregate score for d with respect to terms in T_i. Although there are several possibilities for the monotonic aggregate function to be used, we employ summation, for simplicity. Hence, $TotalScore(d) = \sum_{t \in T_i} score(d, t)$. For a given term, $RankKscore$ refers to the k-th highest TotalScore, s_{min} (s_{max}) refers to the minimum (maximum) score value, and, given a score s, $next(s)$ ($prev(s)$) refers to the score value immediately following (preceding) s.

All nodes are connected on a *global* network G. G is an *overlay network*, modeled as a graph $G = (N, E)$, where E denotes the communication links connecting the nodes. E is explicitly defined by the choice of the overlay network; for instance, for Chord, E consists of the successor, predecessor, and finger table (i.e., routing table) links of each node.

In addition to the global network G, encompassing all nodes, our model employs term-specific overlays, coined *Term Index Networks (TINs)*. $I(t)$ denotes the TIN for term t and is used to store and maintain all (t, d, s) items. TIN $I(t)$ is defined as $I(t) = (N(t), E(t))$, $N(t) \subseteq N$. Note that nodes in $N(t)$ have in addition to the links for participating in G, links needed to connect them to the $I(t)$ network. The model itself is independent of any particular overlay architecture.

$I(t).n(s_i)$ defines the node responsible for storing all triplets (t, d, s) for which $score(d, t) = s = s_i$. When the context is well understood, the same node is simply denoted as $n(s)$.

4 Design Overview and Rationale

The fundamental distinguishing feature of MINERVA∞ is its high distribution both in the data and computational dimensions. MINERVA∞ goes far beyond the state of the art in distributed top-k query processing algorithms, which are based on having nodes storing complete index lists for terms and running coordinator-based top-k algorithms [2,12]. From a data point of view, the principle is that the data items needed by top-k queries are the triplets $(term, docID, score)$ for each queried term (and not the index lists containing them). A proper distributed design for such systems then should appropriately distribute these items controllably so to meet the goals of scalability and efficiency. Thus, data distribution in MINERVA∞ is at the level of this, much finer data grain. From a system's point of view, the design principle we follow is to organize the key computations to engage several different nodes, with each node having to perform small (sub)tasks, as opposed to assigning single large task to a single node. These design choices, we believe, will greatly boost scalability (especially under skewed accesses).

Our approach to materializing this design relies on the employment of the novel notion of Term Index Networks (TINs). TINs may be formed for every term in our system, and they serve two roles: First, as an abstraction, encapsulating the information specific to a term of interest, and second, as a physical manifestation of a distributed repository of the term-specific data items, facilitating their efficient and scalable retrieval. A TIN can be conceptualized as a *virtual* node storing a *virtually global* index list for a term, which is constructed by the sorted merging of the separate complete index lists for the term computed at different nodes. Thus, TINs are comprised of nodes which collectively store different horizontal partitions of this global index list. In practice, we expect TINs to be employed only for the most popular terms (a few hundred to a few thousand) whose accesses are expected to form scalability and performance bottlenecks.

We will exploit the underlying network $G's$ architecture and related algorithms (e.g., for routing/lookup) to efficiently and scalably create and maintain TINs and for retrieving TIN data items, from any node of G. In general, TINs may form separate overlay networks, coexisting with the global overlay G[1].

The MINERVA∞ algorithms are heavily influenced by the way the well-known, efficient top-k query processing algorithms (e.g., [6]) operate, looking for docIDs within certain ranges of score values. Thus, the networks' *lookup*(s) function, will be used using scores s as input, to locate the nodes storing docIDs with scores s.

A key point to stress here, however, is that top-k queries $Q(\{t_1, ..., t_r\}, k)$ can originate from any peer node p of G, which in general is not a member of any $I(t_i)$, $i = 1, ..., r$ and thus p does not have, nor can it easily acquire, the necessary routing state needed to forward the query to the TINs for the query terms. Our infrastructure, solves this by utilizing for each TIN a fairly small number (relative to the total number of data items for a term) of nodes of G

[1] In practice, it may not always be necessary or advisable to form full-fledged separate overlays for TINs; instead, TINs will be formed as straightforward extensions of G: in this case, when a node n of G joins a TIN, only two additional links are added to the state of n linking it to its successor and predecessor nodes in the TIN. In this case, a TIN is simply a (circular) doubly-linked list.

which will be readily identifiable and accessible from any node of G and can act as *gateways* between G and this TIN, being members of both networks.

Finally, in order for any highly distributed solution to be efficient, it is crucial to keep as low as possible the time and bandwidth overheads involved in the required communication between the various nodes. This is particularly challenging for solutions built over very large scale infrastructures. To achieve this, the algorithms of MINERVA∞ follow the principles put forward by top-performing, resource-efficient top-k query processing algorithms in traditional environments. Specifically, the principles behind favoring sequential index-list accesses to random accesses (in order to avoid high-cost random disk IOs) have been adapted in our distributed algorithms to ensure that: (i) sequential accesses of the items in the *global, virtual* index list dominate, (ii) they require either no communication, or at most an one-hop communication between nodes, and (iii) random accesses require at most $O(log|N|)$ messages.

To ensure the at-most-one-hop communication requirement for successive sequential accesses of TIN data, the MINERVA∞ algorithms utilize an *order preserving hash function*, first proposed for supporting range queries in DHT-based data networks in [20]. An order preserving hash function $h_{op}()$ has the property that for any two values $v_1,\ v_2$, if $v_1 > v_2$ then $h_{op}(v_1) > h_{op}(v_2)$. This guarantees that data items corresponding to successive score values of a term t are placed either at the same or at neighboring nodes of $I(t)$. Alternatively, similar functionality can be provided by employing for each $I(t)$ an overlay based on skip graphs or skip nets [1,9]. Since both order preserving hashing and skip graphs incur the danger for load imbalances when assigning data items to TIN nodes, given the expected data skew of scores, load balancing solutions are needed.

The design outlined so far, leverages DHT technology to facilitate efficiency and scalability in key aspects of the system's operation. Specifically, posting (and deleting) data items for a term from any node can be done in $O(log|N|)$ time, in terms of the number of messages. Similarly, during top-k query processing, the TINs of the terms in the query can be also reached in $O(log|N|)$ messages. Furthermore, no single node is over-burdened with tasks which can either require more resources than available, or exhaust its resources, or even stress its resources for longer periods of time. In addition, as the top-k algorithm is processing different data items for each queried term, this involves gradually different nodes from each TIN, producing a highly distributed, scalable solution.

5 Term Index Networks

In this section we describe and analyze the algorithms for creating TINs and populating them with data and nodes.

5.1 Beacons for Bootstrapping TINs

The creation of a TIN has these basic elements: posting data items, inserting nodes, and maintaining the connectivity of nodes to ensure the efficiency/scalability properties promised by the TIN overlay.

As mentioned, a key issue to note is that any node p in G may need to post (t, d, s) items for a term t. Since, in general, p is not a member of $I(t)$ and does not necessarily know members of $I(t)$, efficiently and scalably posting items to $I(t)$ from any p becomes non-trivial. To overcome this, a bootstrapping process for $I(t)$ is employed which initializes an TIN $I(t)$ for term t. The basic novelty lies in the special role to be played by nodes coined *beacons*, which in essence become gateways, allowing the flow of data and requests between the G and $I(t)$ networks.

In the bootstrap algorithm, a predefined number of "dummy" items of the form $(t, \star, s_i)$ is generated in sequence for a set of predefined score values s_i, $i = 1, ..., u$. Each such item will be associated with a node n in G, where it will be stored. Finally, this node n of G will also be made a member of $I(t)$ by randomly choosing a previously inserted beacon node (i.e., for the one associated with an already inserted score value s_j, $1 \leq j \leq i - 1$) as a gateway.

The following algorithm details the pseudocode for bootstrapping $I(t)$. It utilizes an order-preserving hash function $h_{op}() : T \times (0, 1] \rightarrow [m]$, where m is the size of the identifiers in bits and $[m]$ denotes the name space used for the overlay (e.g., all 2^{160} ids, for 160-bit identifiers). In addition, a standard hash function $h() : (0, 1] \rightarrow [m]$, (e.g. SHA-1) is used. The particulars of the order preserving hash function to be employed will be detailed after the presentation of the query processing algorithms which they affect. The bootstrap algorithm selects u "dummy" score values, i/u, $i = 1, ..., u$, finds for each such score value the node n in G where it should be placed (using $h_{op}()$), stores this score there and inserts n into the $I(t)$ network as well. At first, the $I(t)$ network contains only the node with the dummy item with score zero. At each iteration, another node of n is added to $I(t)$ using as gateway the node of G which was added in the previous iteration to $I(t)$. For simplicity of presentation, the latter node

Algorithm 1. Bootstrap $I(t)$

```
input: $u$: the number of "dummy" items $(t, \star, s_i)$, $i = 1, ..., u$
input: $t$: the term for which the TIN is created
$p = 1/u$
for $i = 1$ to $u$ do
    $s = i \times p$
    $lookup(n.s) = h_{op}(t, s)$ { $n.s$ in $G$ will become the next beacon node of $I(t)$ }
    if $s = p$ then
        $N(t) = \{n.s\}$
        $E(t) = \emptyset$ {Initialized $I(t)$ with $n.s$ with the first dummy item}
    end if
    if $s \neq p$ then
        $n_1 = h_{op}(t, s - p)$ {insert $n(s)$ into $I(t)$ using node $n(s - p)$ as gateway}
        call $join(I(t), n_1, s)$
    end if
    store $(t, \star, s)$ at $I(t).n(s)$
end for
```

can be found by simply hashing for the previous dummy value. A better choice for distributing the load among the beacons is to select at random one of the previously-inserted beacons and use it as a gateway.

Obviously, a single beacon per TIN suffices. The number u of beacon scores is intended to introduce a number of gateways between G and $I(t)$ so to avoid potential bottlenecks during TIN creation. u will typically be a fairly small number so the total beacon-related overhead involved in the TIN creation will be kept small. Further, we emphasize that beacons are utilized by the algorithm posting items to TINs. Post operations will in general be very rare compared to query operations and query processing does not involve the use of beacons.

Finally, note that the algorithm uses a $join()$ routine that adds a node $n(s)$ storing score s into $I(t)$ using a node n_1 known to be in $I(t)$ and thus, has the required state. The new node $n(s)$ must occupy a position in $I(t)$ specified by the value of $h_{op}(t, s)$. Note that this is ensured by using $h(nodeID)$, as is typically done in DHTs, since these node IDs were selected from the order-preserving hash function. Besides the side-effect of ensuring the order-preserving position for the nodes added to a TIN, the join routine is otherwise straightforward: if the TIN is a full-fledged DHT overlay, join() is updating the predecessor/successor pointers, the $O(log|N|)$ routing state of the new node, and the routing state of each $I(t)$ node pointing to it, as dictated by the relevant DHT algorithm. If the TIN is simply a doubly-linked list, then only predecessor/successor pointers are the new node and its neighbors are adjusted.

5.2 Posting Data to TINs

The posting of data items is now made possible using the bootstrapped TINs. Any node n_1 of G wishing to post an item (t, d, s) first locates an appropriate node of G, n_2 that will store this item. Subsequently, it inserts node n_2 into $I(t)$. To do this, it randomly selects a beacon score and associated beacon node, from all available beacons. This is straightforward given the predefined beacon score values and the hashing functions used. The chosen beacon node has been made a member of $I(t)$ during bootstrapping. Thus, it can “escort” n_2 into $I(t)$.

The following provides the pseudocode for the posting algorithm. By design, the post algorithm results in a data placement which introduces two characteristics, that will be crucial in ensuring efficient query processing. First, (as the bootstrap algorithm does) the post algorithm utilizes the order-preserving hash function. As a result, any two data items with consecutive score values for the same term will be placed by definition in nodes of G which will become one-hop neighbors in the TIN for the term, using the join() function explained earlier. Note, that within each TIN, there are no ‘holes’. A node n becomes a member of a TIN network if and only if a data item was posted, with the score value for this item hashing to n. It is instructing here to emphasize that if TINs were not formed and instead only the global network was present, in general, any two successive score values could be falling in nodes which in G could be many hops apart. With TINs, following successor (or predecessor) links always leads to

Algorithm 2. Posting Data to $I(t)$

```
1: input: $t, d, s$: the item to be inserted by a node $n_1$
2: $n(s) = h_{op}(t, s)$
3: $n_1$ sends $(t, d, s)$ to $n(s)$
4: if $n(s) \notin N(t)$ then
5:    $n(s)$ selects randomly a beacon score $s_b$
6:    $lookup(n_b) = h_{op}(t, s_b)$ { $n_b$ is the beacon node storing beacon score $s_b$ }
7:    $n(s)$ calls $join(I(t), n_b, s)$
8: end if
9: store $((t, d, s)$
```

nodes where the next (or previous) segment of scores have been placed. This feature in essence ensures the at-most-one-hop communication requirement when accessing items with successive scores in the global virtual index list for a term.

Second, the nodes of any $I(t)$ become responsible for storing specific segments (horizontal partitions) of the global virtual index list for t. In particular, an $I(t)$ node stores all items for t for a specific (range of) score value, posted by any node of the underlying network G.

5.3 Complexity Analysis

The bootstrapping $I(t)$ algorithm is responsible for inserting u beacon items. For each beacon item score, the node $n.s$ is located by applying the $h_{op}()$ function and routing the request to that node (step 5). This will be done using G's lookup algorithm in $O(log|N|)$ messages. The next key step is to locate the previously inserted beacon node (step 11) (or any beacon node at random) and sending it the request to join the TIN. Step 11 again involves $O(log|N|)$ messages. The actual join() routine will cost $O(log^2|N(t)|)$ messages, which is the standard join() message complexity for any DHT of size $N(t)$. Therefore, the total cost is $O(u \times (log|N| + log^2|N(t)|)$ messages.

The analysis for the posting algorithm is very similar. For each $post(t, d, s)$ operation, the node n where this data item should be stored is located and the request is routed to it, costing $O(log|N|)$ messages (step 3). Then a random beacon node is located, costing $O(log|N|)$ messages, and then the join() routine is called from this node, costing $O(log^2|N(t)|)$ messages. Thus, each post operation has a complexity of $O(log|N|) + O(log^2|N(t)|)$ messages.

Note that both of the above analysis assumed that each $I(t)$ is a full-blown DHT overlay. This permits a node to randomly select any beacon node to use to join the TIN. Alternatively, if each $I(t)$ is simply a (circular) doubly-linked list, then a node can join a TIN using the beacon storing the beacon value that is immediately preceding the posted score value. This requires $O(log|N|)$ hops to locate this beacon node. However, since in this case the routing state for each node of a TIN consists of only the two (predecessor and successor) links, the cost to join is in the worst case $O(|N(t)|)$, since after locating the beacon node with the previous beacon value, $O(|N(t)|)$ successor pointers may need to be followed in order to place the node in its proper order-preserving position. Thus, when

TINs are simple doubly-linked lists, the complexity of both the bootstrap and post algorithms are $O(log|N| + |N(t)|)$ messages.

6 Load Balancing

6.1 Order-Preserving Hashing

The order preserving hash function to be employed is important for several reasons. First, for simplicity, the function can be based on a simple linear transform. Consider hashing a value $f(s) : (0, 1] \rightarrow I$, where $f(s)$ transforms a score s into an integer; for instance, $f(s) = 10^6 \times s$. Function $h_{op}()$ can be defined then as

$$h_{op}(s) = (\frac{f(s) - f(s_{min})}{f(s_{max}) - f(s_{min})} \times 2^m)\ mod\ 2^m \tag{1}$$

Although such a function is clearly order-preserving, it has the drawback that it produces the same output for items of equal scores of different terms. This leads to the same node storing for all terms all items having the same score. This is undesirable since it cannot utilize all available resources (i.e., utilize different sets of nodes to store items for different terms). To avoid this, $h_{op}()$ is refined to take as input the term name, which provides the necessary functionality, as follows.

$$h_{op}(t, s) = (h(t) + \frac{f(s) - f(s_{min})}{f(s_{max}) - f(s_{min})} \times 2^m)\ mod\ 2^m \tag{2}$$

The term $h(t)$ adds a different random offset for different terms, initiating the search for positions of term score values at different, random, offsets within the namespace. Thus, by using the $h(t)$ term in $h_{op}(t, s)$ the result is that any data items having equal scores but for different terms are expected to be stored at different nodes of G.

Another benefit stems from ameliorating the storage load imbalances that result from the non-uniform distribution of score values. Assuming a uniform placement of nodes in G, the expected non-uniform distribution of scores will result in a non-uniform assignment of scores to nodes. Thus, when viewed from the perspective of a single term t, the nodes of $I(t)$ will exhibit possibly severe storage load imbalances. However, assuming the existence of large numbers of terms (e.g., a few thousand), and thus data items being posted for all these terms over the same set of nodes in G, given the randomly selected starting offsets for the placement of items, it is expected that the severe load imbalances will disappear. Intuitively, overburdened nodes for the items of one term are expected to be less burdened for the items of other terms and vice versa.

But even with the above hash function, very skewed score distributions will lead to storage load imbalances. Expecting that exponential-like distributions of score values will appear frequently, we developed a hash function that is order-preserving and handles load imbalances by assigning score segments of exponentially decreasing sizes to an exponentially increasing number of nodes. For instance, the sparse top 1/2 of the scores distribution is to be assigned to a single node, the next 1/4 of scores is to be assigned to 2 nodes, the next 1/8 of scores to 4 nodes, etc. The details of this are omitted for space reasons.

6.2 TIN Data Migration

Exploiting the key characteristics of our data, MINERVA∞ can ensure further load balancing with small overheads. Specifically, index lists data entries are small in size and are very rarely posted and/or updated. In this subsection we outline our approach for improved load balancing.

We require that each peer posting index list entries, first computes a (equi-width) histogram of its data with respect to its score distribution. Assuming a targeted $|N(t)|$ number of nodes for the TIN of term t, it can create $|N(t)|$ equal-size partitions, with $lowscore_i, highscore_i$ denoting the score ranges associated with partition i, $i = 1, ..., |N(t)|$. Then it can simply utilize the posting algorithm shown earlier, posting using the $lowscore_i$ scores for each partition. The only exception to the previously shown post algorithm is that the posting peer now posts at each iteration a complete partition of its index list, instead of just a single entry.

The above obviously can guarantee perfect load balancing. However, subsequent postings (typically by other peers) may create imbalances, since different index lists may have different score distributions. Additionally, when ensuring overall load balancing over multiple index lists being posting by several peers, the order-preserving property of the placement must be guaranteed. Our approach for solving these problems is as follows. First, again the posting peer is required to compute a histogram of its index list. Second, the histogram of the TIN data (that is, the entries already posted) is stored at easily identifiable nodes. Third, the posting peer is required to retrieve this histogram and 'merge' it with his own. Fourth, the same peer identifies how the total data must now be split into $|N(t)|$, equal-size partitions of consecutive scores. Finally, it identifies all data movements (from TIN peer to TIN peer) necessary to redistribute the total TIN data so that load balancing and order preservation is ensured.

Detailed presentation of the possible algorithms for this last step and their respective comparison is beyond the scope of this paper. We simply mention that total TIN data sizes is expected to be very small (in actual number of bytes stored and moved). For example, even with several dozens of peers posting different, even large, multi-million-entry, index lists, in total the complete TIN data size will be a few hundred MBs, creating a total data transfer movement equivalent to that of downloading a few dozen MP3 files. Further, index lists' data posting to TINs is expected to be a very infrequent operation (compared to search queries). As a result, ensuring load balancing across TIN nodes proves to be relative inexpensive.

6.3 Discussion

The approaches to index lists' data posting outlined in the previous two sections can be used competitively or even be combined. When posting index lists with exponential score distributions, by design the posting of data using the order-preserving hash function of Section 5.1, will be adequately load balanced and nothing else is required. Conversely, when histogram information is available and can be computed by posting peers, the TIN data migration approach will yield load balanced data placement.

A more subtle issue is that posting with the order-preserving hash function also facilitates random accesses of the TIN data, based on random score values. That is, by hashing for any score, we can find the TIN node holding the entries with this score. This becomes essential if the web search engine is to employ top-k query algorithms which are based on random accesses of scores. In this work, our top-k algorithms avoid random accesses, by design. However, the above point should be kept in mind since there are recently-proposed distributed top-k algorithms, relying on random accesses and more efficient algorithms may be proposed in the future.

7 Top-k Query Processing

The algorithms in this section focus on how to exploit the infrastructure presented previously in order to efficiently process top-k queries. The main efficiency metrics are query response times and network bandwidth requirements.

7.1 The Basic Algorithm

Consider a top-k query of the form $Q(\{t_1, ..., t_r\}, k)$ involving r terms that is generated at some node n_{init} of G. Query processing is based on the following ideas. It proceeds in phases, with each phase involving 'vertical' and 'horizontal' communication between the nodes within TINs and across TINs, respectively. The vertical communications between the nodes of a TIN are occuring in parallel across all r TINs named in the query, gathering a threshold number of data items from each term. There is a moving coordinator node, that will be gathering the data items from all r TINs that enable it to compute estimates of the top-k result. Intermediate estimates of the top-k list will be passed around, as the coordinator role moves from node to node in the next phase where the gathering of more data items and the computation of the next top-k result estimate will be computed.

The presentation shows separately the behavior of the query initiator, the (moving) query coordinator, and the TIN nodes.

Query Initiator

The initiator calculates the set of *start nodes*, one for each term, where the query processing will start within each TIN. Also, it randomly selects one of the nodes (for one of the TINs) to be the initial coordinator. Finally, it passes on the query and the coordinator ID to each of the start nodes, to initiate the parallel vertical processing within TINs.

The following pseudocode details the behavior of the initiator.

Processing Within Each TIN

Processing within a TIN is always initiated by the start node. There is one start node per communication phase of the query processing. In the first phase, the

Algorithm 3. Top-k QP: Query Initiation at node $G.n_{init}$

```
input: Given query $Q = \{t_1,..,t_r\}, k$ :
for $i = 1$ to $r$ do
    $startNode_i = I(t_i).n(s_{max}) = h_{op}(t_i, s_{max})$
end for
Randomly select $c$ from $[1, ..., r]$
$coordID = I(t_c).n(s_{max})$
for $i = 1$ to $r$ do
    send to $startNode_i$ the data $(Q, coordID)$
end for
```

start node is the top node in the TIN which receives the query processing request from the initiator. The start node then starts the gathering of data items for the term by contacting enough nodes, following successor links, until a threshold number γ (that is, a batch size) of items has been accumulated and sent to the coordinator, along with an indication of the maximum score for this term which has not been collected yet, which is actually either a locally stored score or the maximum score of the next successor node. The latter information is critical for the coordinator in order to intelligently decide when the top-k result list has been computed and terminate the search. In addition, each start node sends to the coordinator the ID of the node of this TIN to be the next start node, which is simply the next successor node of the last accessed node of the TIN. Processing within this TIN will be continued at the new start node when it receives the next message from the coordinator starting the next data-gathering phase.

Algorithm 4 presents the pseudocode for TIN processing.

Algorithm 4. Top-k QP: Processing by a start node within a TIN

```
input: A message either from the initiator or the coordinator
$tCollection_i = \emptyset$
$n = startNode_i$
while $|tCollection_i| < \gamma$ do
    while $|tCollection_i| < \gamma$ AND more items exist locally do
        define the set of local items $L = \{(t_i, d, s)$ in $n\}$
        send to $coordID : L$
        $|tCollection_i| = |tCollection_i| + |L|$
    end while
    n = succ(n)
end while
$bound_i$ = max score stored at node $n$
send to $coordID$ : $n$ and $bound_i$
```

Recall, that because of the manner with which items and nodes have been placed in a TIN, following succ() links, items are collected starting from the item with the highest score posted for this term and proceeding in sorted descending order based on scores.

Moving Query Coordinator

Initially, the coordinator is randomly chosen by the initiator to be one of the original start nodes. First, the coordinator uses the received collections and runs a version of the *NRA* top-k processing algorithm, locally producing an estimate of the top-k result. As is also the case with classical top-k algorithms, the exact result is not available at this stage since only a portion of the required information is available. Specifically, some documents with high enough TotalScore to qualify for the top-k result are still missing. Additionally, some documents may also be seen in only a subset of the collections received from the TINs so far, and thus some of their scores are missing, yielding only a partially known TotalScore.

A key to the efficiency of the overall query processing process is the ability to prune the search and terminate the algorithm even in the presence of missing documents and missing scores. To do this, the coordinator first computes an estimate of the top-k result, which includes only documents whose TotalScores are completely known, defining the RankKscore value (i.e. the smallest score in the top-k list estimate). Then, it utilizes the $bound_i$ values received from each start node. When a score for a document d is missing for term i, it can be replaced with $bound_i$ to estimate the $TotalScore(d)$. This is done for all such d with missing scores. If $RankKscore > TotalScore(d)$ for all d with missing scores then there is no need to continue the process for finding the missing scores, since the associated documents could never belong to the top-k result. Similarly, if $RankKscore > \sum_{i=1,\ldots,r} bound_i$, then similarly there is no need to try to find any other documents, since they could never belong to the top-k result. When both of these conditions hold, the coordinator terminates the query processing and returns the top-k result to the initiator.

If the processing must continue, the coordinator starts the next phase, sending a message to the new start node for each term, whose ID was received in the message containing the previous data collections. In this message the coordinator also indicates the ID of the node which becomes the coordinator in this next phase. The next coordinator is defined to be the node in the same TIN as the previous coordinator whose data is to be collected next in the vertical processing in this TIN (i.e., the next start node at the coordinator's TIN). Alternatively, any other start node can be randomly chosen as the coordinator.

Algorithm 5 details the behavior of the coordinator.

7.2 Complexity Analysis

The overall complexity has three main components: the cost incurred for (i) the communication between the query initiator and the start nodes of the TINs, (ii) the vertical communication within a TIN, and (iii) the horizontal communication between the current coordinator and the current set of start nodes.

The query initiator needs to lookup the identity of the initial start nodes for each one of the r query terms and route to them the query and the chosen coordinator ID. Using the G network, this incurs a communication complexity of $O(r \times log|N|)$ messages. Denoting with *depth* the average (or maximum) number

Algorithm 5. Top-k QP: Coordination

```
input: For each $i$: $tCollection_i$ and $newstartNode_i$ and $bound_i$
$tCollection = \bigcup_i tCollection_i$
compute a (new) top-$k$ list estimate using $tCollection$, and RankKscore
$candidates = \{d | d \notin$top-$k$ list$\}$
for all $d \in candidates$ do
  $worstScore(d)$ is the partial TotalScore of $d$
  $bestScore(d) := worstScore(d) + \sum_{j \in MT} bound_j$ {Where $MT$ is the set of term
  ids with missing scores }
  if $bestScore(d) < RankKscore$ then
    remove $d$ from $candidates$
  end if
end for
if $candidates$ is empty then
  exit()
end if
if $candidates$ is not empty then
  $coordID_{new} = pred(n)$
  calculate new size threshold $\gamma$
  for $i = 1$ to $r$ do
    send to $startNode_i$ the data $(coordID_{new}, \gamma)$
  end for
end if
```

of nodes accessed during the vertical processing of TINs, overall $O(r \times depth)$ messages are incurred due to TIN processing, since subsequent accesses within a TIN require, by design, one-hop communication. Each horizontal communication in each phase of query processing between the coordinator and the r start nodes requires $O(r \times log|N|)$ messages. Since such horizontal communication takes place at every phase, this yields a total of $O(phases \times r \times log|N|)$ messages. Hence, the total communication cost complexity is

$$cost = O(phases \times r \times log|N| + r \times log|N| + r \times depth) \qquad (3)$$

This total cost is the worst case cost; we expect that the cost incurred in most cases will be much smaller, since horizontal communication across TINs can be much more efficient than $O(log|N|)$, as follows. The query initiator can first resolve the ID of the coordinator (by hashing and routing over G) and then determine its actual physical address (i.e., its IP address), which is then forwarded to each start node. In turn, each start node can forward this from successor to successor in its TIN. In this way, at any phase of query processing, the last node of a TIN visited during the vertical processing, can send the data collection to the coordinator using the coordinator's physical address. The current coordinator also knows the physical address of the next coordinator (since this was the last node visited in its own TIN from which it received a message with the data collection for its term) and of the next start node for all terms (since these are the last nodes visited during vertical processing of the TINs, from which it received a message). Thus, when sending the message to the next start nodes to continue vertical processing, the physical addresses can be used. The end result of this is that all horizontal communication requires one message, instead of $O(log|N|)$ messages. Hence, the total communication cost complexity now becomes

$$cost = O(phases \times r + r \times log|N| + r \times depth) \tag{4}$$

As nodes are expected to be joining and leaving the underlying overlay network G, occasionally, the physical addresses used to derive the cost of (4) will not be valid. In this case, the reported errors will lead to nodes using the high-level IDs instead of the physical addresses, in which case the cost is that given by (3).

8 Expediting Top-k Query Processing

In this section we develop optimizations that can further speedup the performance of top-k query processing. These optimizations are centered on: (i) the 'vertical' replication of term-specific data among the nodes of a TIN, and (ii) the 'horizontal' replication of data across TINs.

8.1 TIN Data Replication

There are two key characteristics of the data items in our model, which permit their large-scale replication. First, data items are rarely posted and even more rarely updated. Second, data items are very small in size (e.g. $<$ 50 bytes each). Hence, replication protocols will not cost significantly either in terms of replica state maintenance, or in terms of storing the replicas.

Vertical Data Replication. The issue to be addressed here is how to appropriately replicate term data within TIN peers so to gain in efficiency. The basic structure of the query processing algorithm presented earlier facilitates the easy incorporation of a replication protocol into it. Recall, that in each TIN $I(t)$, query processing proceeds in phases, and in each phase a TIN node (the current start node) is responsible for visiting a number of other TIN nodes, a successor at a time, so that enough, (i.e., a batch size of) data items for t are collected. The last visited node in each phase which collects all data items, can initiate a 'reverse' vertical communication, in parallel to sending the collection to the coordinator. With this reverse vertical communication thread, each node in the reverse path sends to its predecessor only the data items its has not seen. In the end, all nodes in the path from the start node to the last node visited will eventually receive a copy of all items collected during this phase, storing locally the pair $(lowestscore, highestscore)$, marking its lowest and highest locally stored scores. Since this is straightforward, the pseudocode is omitted for space reasons.

Since a new posting involves all (or most) of the nodes in these paths, each node knows when to initiate a new replication to account for the new items.

Exploiting Replicas. The start node selected by the query initiator no longer needs to perform a successor-at-a-time traversal of TIN in the first phase, since the needed data (replicas) are stored locally. However, vertical communication was also useful for producing the ID of the next start node for this TIN. A subtle point to note here is that the coordinator can itself determine the new start node for the next phase, even without receiving explicitly this ID at the end of vertical communication. This can simply be done using the minimum score value ($bound_i$) it has received for term t_i; the ID of the next start node is found hashing for score $prev(bound_i)$.

Additionally, the query initiator can select as start nodes the nodes responsible for storing a random (expected to be high score) and not always the maximum score, as it does up to now. Similarly, the coordinator when selecting the ID of the next start node for the next batch retrieval for a term, it can choose to hash for a score value that is lower than the score $prev(bound_i)$. Thus, random start nodes within a TIN are selected at different phases and these gather the next batch of data from the proper TIN nodes, using the TIN DHT infrastructure for efficiency. The details of how this is done, are omitted for space reasons.

Horizontal Data Replication. TIN data may also be replicated horizontally. The simplest strategy is to create replicated TINs for popular terms. This involves the posting of data into all TIN replicas. The same algorithms can be used as before for posting, except now when hashing, instead of using the term t as input to the hash function, each replica of t must be specified (e.g., $t.v$, where v stands for a version/replica number). Again, the same algorithms can be used for processing queries, with the exception that each query can now select one of the replicas of $I(t)$, at random.

Overall, TIN data replication leads to savings in the number of messages and response time speedups. Furthermore, several nodes are off-loaded since they no longer have to partake in the query processing process. With replication, therefore, the same number of nodes overall will be involved in processing a number of user queries, except that each query will be employing a smaller set of peers, yielding response time and bandwidth benefits. In essence, TIN data replication increases the efficiency of the engine, without adversely affecting its scalability. Finally, it should be stressed that such replication will also improve the availability of data items and thus replication is imperative. Indirectly, for the same reason the quality of the results with replication will be higher, since lost items inevitably lead to errors in the top-k result.

9 Experimentation

9.1 Experimental Testbed

Our implementation was written in Java. Experiments were performed on 3GHz Pentium PCs. Since deploying full-blown, large networks is not an option, we opted for simulating large numbers of nodes as separate processes on the same PC, executing the real MINERVA∞ code. A 10,000 node network was simulated.

A real-world data collection was used in our experiments: GOV. The GOV collection consists of the data of the TREC-12 Web Track and contains roughly 1.25 million (mostly HTML and PDF) documents obtained from a crawl of the .gov Internet domain (with total index list size of 8 GB). The original 50 queries from the Web Track's distillation task were used. These are term queries, with each query containing up to 4 terms. The index lists contained the original document scores computed as tf * log idf. tf and idf were normalized by the maximum tf value of each document and the maximum idf value in the corpus, respectively. In addition, we employed an extended GOV (XGOV) setup, with a larger number of query terms and associated index lists. The original 50 queries were expanded by adding new terms from synonyms and glosses taken from the WordNet thesaurus (http://www.cogsci.princeton.edu/∼wn). The expansion yielded queries with, on average, twice as many terms, up to 18 terms.

9.2 Performance Tests and Metrics

Efficiency Experiments. The data (index list entries) for the terms to be queried were first posted. Then, the GOV/XGOV benchmark queries were executed in sequence. For simplicity, the query initiator node assumed the role of a fixed coordinator. The experiments used the following metrics:

Bandwidth. This shows the number of bytes transferred between all the nodes involved in processing the benchmarks' queries. The benchmarks' queries were grouped based on the number of terms they involved. In essence, this grouping created a number of smaller sub-benchmarks.

Query Response Time. This represents the elapsed, "wall-clock" time for running the benchmark queries. We report on the wall-clock times per sub-benchmark and for the whole GOV and XGOV benchmarks.

Hops. This reports the number of messages sent over our network infrastructures to process all queries. For communication over the global DHT G, the number of hops was set to be $log|N|$ (i.e., when the query initiator contacts the first set of start nodes for each TIN). Communication between peers within a TIN requires, by design, one hop at a time.

To avoid the overestimation of response times due to the competition between all processes for the PC's disk and network resources, and in order to produce reproducible and comparable results for tests ran at different times, we opted for simulating disk IO latency and network latency. Specifically, each random disk IO was modeled to incur a disk seek and rotational latency of 9 ms, plus a transfer delay dictated by a transfer rate of 8MB/s. For network latency we utilized typical round trip times (RTTs) of packets and transfer rates achieved for larger data transfers between widely distributed entities [16]. We assumed a RTT of 100 ms. When peers simply forward the query to a next peer, this is assumed to take roughly 1/3 of the RTT (since no ACKs are expected). When peers sent more data, the additional latency was dictated by a "large" data transfer rate of 800Kb/s, which includes the sender's uplink bandwidth, the receivers downlink bandwidth, and the average internet bandwidth typically witnessed.[2]

Scalability Experiments. The tested scenarios varied the query load to the system, measuring the overall time required to complete the processing of all queries in a queue of requests. Our experiments used a queue of identical queries involving four terms, with varying index lists characteristics. Two of these terms had small index lists (with over 22,000 and over 42,000 entries) and the other two lists had sizes of over 420,000 entries. For each query the (different) query initiating peer played the role of the coordinator.

The key here is to measure contention for resources and its limits on the possible parallelization of query processing. Each TIN peer uses his disk, his uplink bandwidth to forward the query to his TIN successor, and to send data to the coordinator. Uplink/downlink bandwidths were set to 256Kbps/1Mbps. Similarly, the query initiator utilizes its downlink bandwidth to receive the batches of data in each phase and its uplink bandwidth to send off the query to the next TIN start nodes. These delays define the possible parallelization of query execution. By involving the two terms with the largest index lists in the queries, we ensured the worst possible parallelization (for our input data), since they induced the largest batch size, requiring the most expensive disk reads and communication.

9.3 Performance Results

Overall, each benchmark experiment required between 2 to 5 hours for its real-time execution, a big portion of which was used up by the posting procedure.

Figures 1 and 2 show the bandwidth, response times, and hops results for the GOV and XGOV group-query benchmarks. Note, that different query groups have in general mutually-incomparable results, since they involve different index lists with different characteristics (such as size, score distributions etc).

In XGOV the biggest overhead was introduced by the 8 7-term and 6 11-term queries. Table 1 shows the total benchmark execution times, network bandwidth consumption, as well as the number of hops for the GOV and XGOV benchmarks.

Generally, for each query, the number of terms and the size of the corresponding index list data are the key factors. The central insight here is that the choice of the NRA algorithm was the most important contributor to the overhead. The adaptation of more efficient distributed top-k algorithms within MINERVA∞ (such as our own [12], which also disallow random accesses) can reduce this overhead by one to two orders of magnitude. This is due to the fact that the top-k result can be produced without needing to delve deeply into the index lists' data, resulting in drastically fewer messages, bandwidth, and time requirements.

[2] This figure is the average throughput value measured (using one stream – one cpu machines) in experiments conducted for measuring wide area network throughput (sending 20MB files between SLAC nodes (Stanford's Linear Accelerator Centre) and nodes in Lyon France [16] using NLANR's iPerf tool [19].

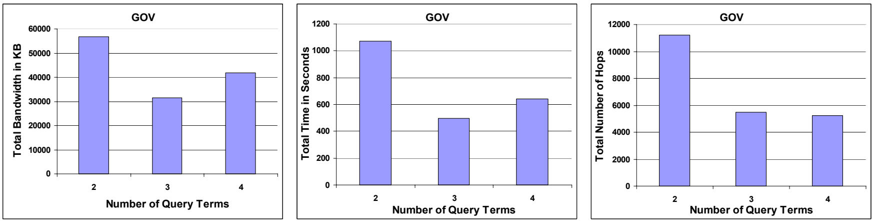

Fig. 1. GOV Results: Bandwidth, Execution Time, and Hops

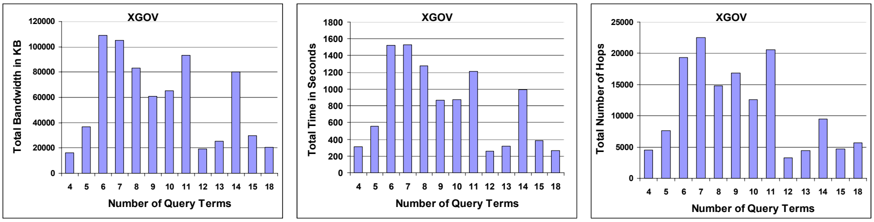

Fig. 2. XGOV Results: Bandwidth, Execution Time, and Hops

Table 1. Total GOV and XGOV Results

Benchmark	Hops	Bandwidth(KB)	Time(s)
GOV	22050	130189	2212
XGOV	146168	744700	10372

The 2-term queries introduced the biggest overheads. There are 29 2-term, 7 3-term, and 4 4-term queries in GOV.

Figure 3 shows the scalability experiment results. Query loads tested represent queue sizes of 10, 100, 1000, and 10000 identical queries simultaneously arriving into the system. This figure also shows what the corresponding time would be if the parallelization contributed by the MINERVA∞ architecture was not possible; this would be the case, for example, in all related-work P2P search architectures and also distributed top-k algorithms, where the complete index lists at least for one query term are stored completely at one peer. The scalability results show the high scalability achievable with MINERVA∞. It is due to the "pipelining" that is introduced within each TIN during query processing, where a query consumes small amounts of resources from each peer, pulling together the resources of all (or most) peers in the TIN for its processing. For comparison we also show the total execution time in an environment in which each complete index list was stored in a peer. This is the case for most related work on P2P search engines and on distributed top-k query algorithms. In this case, the resources of the single peer storing a complete index list are required

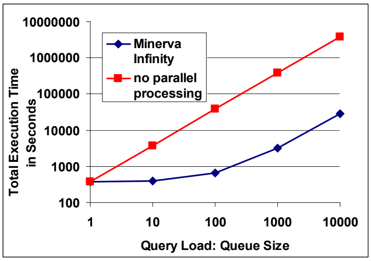

Fig. 3. Scalability Results

for the processing of all communication phases and for all queries in the queue. In essence, this yields a total execution time that is equal to that of a sequential execution of all queries using the resources of the single peers storing the index lists for the query terms. Using this as a base comparison, MINERVA∞ is shown to enjoy approximately two orders of magnitude higher scalability. Since in our experiments there are approximately 100 nodes per TIN, this defines the maximum scalability gain.

10 Concluding Remarks

We have presented MINERVA∞, a novel architecture for a peer-to-peer web search engine. The key distinguishing feature of MINERVA∞ is its high-levels of distribution for both data and processing. The architecture consists of a suite of novel algorithms, which can be classified into algorithms for creating Term Index Networks, TINs, placing index list data on TINs and of top-k algorithms. TIN creation is achieved using a bootstrapping algorithm and also depends on how nodes are selected when index lists data is posted. The data posting algorithm employs an order-preserving hash function and, for higher levels of load balancing, MINERVA∞ engages data migration algorithms. Query processing consists of a framework for highly distributed versions of top-k algorithms, ranging from simple distributed top-k algorithms, to those utilizing vertical and/or horizontal data replication. Collectively, these algorithms ensure efficiency and scalability. Efficiency is ensured through the fast sequential accesses to index lists' data, which requires at most one hop communication and by algorithms exploiting data replicas. Scalability is ensured by engaging a larger number of TIN peers in every query, with each peer being assigned much smaller subtasks, avoiding centralized points of control. We have implemented MINERVA∞ and conducted detailed performance studies showcasing its scalability and efficiency.

Ongoing work includes the adaptation of recent distributed top-k algorithms (e.g., [12]) into the MINERVA∞ architecture, which have proved one to two orders of magnitude more efficient than the NRA top-k algorithm currently employed, in terms of query response times, network bandwidth, and peer loads.

References

1. J. Aspnes and G. Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, Jan. 2003.
2. P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks, PODC 2004.
3. S. Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data.* Morgan Kaufmann, San Francisco, 2002.
4. F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. Technical Report DCS-TR-487, Rutgers University, Sept. 2002.
5. R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.
6. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4), 2003.
7. P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, pages 444–455, 2004.
8. A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc.
9. N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USITS*, 2003.
10. R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, pages 321–332, 2003.
11. J. Lu and J. Callan. Content-based retrieval in hybrid peer-to-peer networks. In *Proceedings of CIKM03*, pages 199–206. ACM Press, 2003.
12. S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *VLDB Conference*, 2005.
13. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM 2001*, pages 161–172. ACM Press, 2001.
14. P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of International Middleware Conference*, pages 21–40, June 2003.
15. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.
16. D. Salomoni and S. Luitz. High performance throughput tuning/measurement. http://www.slac.stanford.edu/grp/scs/net/talk/High_perf_ppdg_jul2000.ppt. 2000.
17. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001*, pages 149–160. ACM Press, 2001.
18. T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasunderam. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. Technical report, Polytechnic Univ., 2003.
19. A. Tirumala et al. iperf: Testing the limits of your network. http://dast.nlanr.net/projects/iperf/. 2003.
20. P. Triantafillou and T. Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In *DBISP2P*, 2003.
21. Y. Wang, L. Galanis, and D. J. de Witt. Galanx: An efficient peer-to-peer search engine system. *Available at http://www.cs.wisc.edu/ yuanwang.*