Top-k Query Processing in the APPA P2P System[1]

Reza Akbarinia[1,3],Vidal Martins[1,2], Esther Pacitti[1], Patrick Valduriez[1]

[1]ATLAS group, INRIA and LINA, University of Nantes, France
[2]PPGIA/PUCPR – Pontifical Catholic University of Paraná, Brazil
[3]Shahid Bahonar University of Kerman, Iran
{FirstName.LastName@univ-nantes.fr, Patrick.Valduriez@inria.fr}

Abstract. Top-k queries are attractive for users in P2P systems with very large numbers of peers but difficult to support efficiently. In this paper, we propose a fully distributed algorithm for executing Top-k queries in the context of the APPA (Atlas Peer-to-Peer Architecture) data management system. APPA has a network-independent architecture that can be implemented over various P2P networks. Our algorithm requires no global information, does not depend on the existence of certain peers and its bandwidth cost is low. We validated our algorithm through implementation over a 64-node cluster and simulation using the BRITE topology generator and SimJava. Our performance evaluation shows that our algorithm has logarithmic scale up and improves Top-k query response time very well using P2P parallelism in comparison with baseline algorithms.

1 Introduction

Peer-to-peer (P2P) systems adopt a completely decentralized approach to data sharing and thus can scale to very large amounts of data and users. Popular examples of P2P systems such as Gnutella [10] and KaZaA [13] have millions of users sharing petabytes of data over the Internet. Initial research on P2P systems has focused on improving the performance of query routing in unstructured systems, such as Gnutella and KaaZa, which rely on flooding. This work led to structured solutions based on distributed hash tables (DHT), *e.g.* CAN [16], or hybrid solutions with super-peers that index subsets of peers [23]. Although these designs can give better performance guarantees than unstructured systems, more research is needed to understand their trade-offs between autonomy, fault-tolerance, scalability, self-organization, etc. Meanwhile, the unstructured model which imposes no constraint on data placement and topology remains the most used today on the Internet

Recently, other work in P2P systems has concentrated on supporting advanced applications which must deal with semantically rich data (*e.g.* XML documents, relational tables, etc.) using a high-level SQL-like query language, *e.g.* ActiveXML [2], Piazza [20], PIER [12]. High-level queries over a large-scale P2P system may produce very large numbers of results that may overwhelm the users. To avoid such overwhelming, a solution is to use Top-k queries whereby the user can specify a

[1] Work partially funded by the ARA Massive Data of the Agence Nationale de la Recherche.

limited number (k) of the most relevant answers. Initial work on Top-k queries has concentrated on SQL-like language extensions [7][6] . In [6] for instance, there is a STOP AFTER k clause to express the k most relevant tuples together with a scoring function to determine their ranking.

Efficient execution of Top-k queries in a large-scale distributed system is difficult. To process a Top-k query, a naïve solution is that the query originator sends the query to all nodes and merges all the results, which it gets back. This solution hurts response time as the central node is a bottleneck and does not scale up. Efficient techniques have been proposed for Top-k query execution in distributed systems [25][24]. They typically use histograms, maintained at a central site, to estimate the score of databases with respect to the query and send the query to the databases that are more likely to involve top results. These techniques can somehow be used in super-peer systems where super-peers maintain the histograms and perform query sending and result merging. However, keeping histograms up-to-date with autonomous peers that may join or leave the system at any time is difficult. Furthermore, super-peers can also be performance bottlenecks. In unstructured or DHT systems, these techniques which rely on central information no longer apply.

In this paper, we propose a fully distributed algorithm for executing Top-k queries processing in the context of APPA (Atlas Peer-to-Peer Architecture), a P2P data management system which we are building [3][4]. The main objectives of APPA are scalability, availability and performance for advanced applications. APPA has a network-independent architecture in terms of advanced services that can be implemented over different P2P networks (unstructured, DHT, super-peer, etc.). This allows us to exploit continuing progress in such systems. Our Top-k query processing algorithm has several distinguishing features. For instance, it requires no central or global information. Furthermore, its execution is completely distributed and does not depend on the existence of certain peers. We validated our algorithm through a combination of implementation and simulation and the performance evaluation shows very good performance. We have also implemented baseline algorithms for comparing with our algorithm. Our performance evaluation shows that our algorithm improves Top-k query response time very well using P2P parallelism in comparison with baseline algorithms.

The rest of this paper is organized as follows. Section 2 describes the APPA architecture. In Section 3, we present our algorithm, then we analyzes the bandwidth cost of our algorithm and propose techniques in order to reduce this cost. Section 4 describes a performance evaluation of the algorithm through implementation over a 64-node cluster and simulation (up to 10,000 peers) using the BRITE topology generator [5] and SimJava [11]. Section 5 discusses related work. Section 6 concludes.

2 APPA Architecture

APPA has a layered service-based architecture. Besides the traditional advantages of using services (encapsulation, reuse, portability, etc.), APPA is a network-independent architecture so it can be implemented over different P2P networks

(unstructured, DHT, super-peer, etc.). The main reason for this choice is to be able to exploit rapid and continuing progress in P2P networks. Another reason is that it is unlikely that a single P2P network design will be able to address the specific requirements of many different applications. Obviously, different implementations will yield different trade-offs between performance, fault-tolerance, scalability, quality of service, etc. For instance, fault-tolerance can be higher in unstructured P2P systems because no peer is a single point of failure. On the other hand, through index servers, super-peer systems enable more efficient query processing. Furthermore, different P2P networks could be combined in order to exploit their relative advantages, *e.g.* DHT for key-based search and super-peer for more complex searching.

There are three layers of services in APPA: P2P network, basic services and advanced services.

P2P network. This layer provides network independence with services that are common to all P2P networks, for instance:

- **Peer id assignment:** assigns a unique id to a peer using a specific method, *e.g.* a combination of super-peer id and counter in a super-peer network.
- **Peer linking:** links a peer to some other peers, *e.g.* by setting neighbors in an unstructured network, by locating a zone in CAN [16], etc. It also maintains the address and id of the peer's neighbors.
- **Peer communication:** enables peers to exchange messages (*i.e.* service calls).

Basic services. This layer provides elementary services for the advanced services using the P2P network layer, for instance:

- **P2P data management:** stores and retrieves P2P data (*e.g.* meta-data, index data) in the P2P network.
- **Peer management:** provides support for peer joining, rejoining, and for updating peer address (the peer ID is permanent but its address may be changed).
- **Group membership management:** allows peers to join an abstract *group*, become *members* of the group and send and receive membership notifications.

Advanced services. This layer provides advanced services for semantically rich data sharing including schema management, replication, query processing, security, etc. using the basic services.

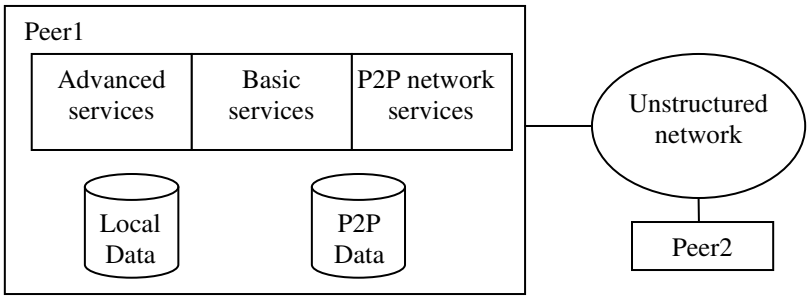

Fig. 1. APPA architecture over an unstructured network

For the cases where APPA is based on a DHT or an unstructured network, the three service layers are completely distributed over all peers, but in a super-peer network the super-peers provide P2P network services and basic services while other peers provide only the advanced services. Figure 1 shows an APPA architecture based on an unstructured network.

3 Top-k Query Processing

In this section, we first make precise our assumptions and define the problem. Then, we present a basic algorithm for Top-k query processing in APPA when it is based on an unstructured P2P system. Finally, we analyze the bandwidth cost of our algorithm and propose some techniques for reducing it.

3.1 Problem Definition

We first give our assumptions regarding schema management and Top-k queries. Then we can precisely state the problem we address in this paper.

In a P2P system, peers should be able to express queries over their own schema without relying on a centralized global schema as in data integration systems [20]. Several solutions have been proposed to support decentralized schema mapping. However, this issue is out of the scope of this paper and we assume it is provided using one of the existing techniques, *e.g.* [15] and [20]. Furthermore, also for simplicity, we assume relational data.

Now we can define the problem as follows. Let Q be a Top-k query, *i.e.* the user is interested to receive k top answers to Q. Let TTL (Time-To-Live) determine the maximum hop distance which the user wants her query be sent. Let D be the set of all data items (*i.e.* tuples) that can be accessed through *ttl* hops in the P2P system during the execution of Q. Let $Sc(d, Q)$ be a scoring function that denotes the score of relevance of a data item $d \in D$ to Q. Our goal is to find the set $T \subseteq D$, such that:

$|T| = k$ and $\forall d_1 \in T, \forall d_2 \in (D - T)$ then $Sc(d_1, Q) \geq Sc(d_2, Q)$

while minimizing the response time of Q and the bandwidth cost.

3.2 Algorithm

The algorithm starts at the *query originator*, the peer at which a user issues a Top-k query Q. The query originator performs some initialization. First, it sets *TTL* with a value which is either specified by the user or default. Second, it gives Q a unique identifier, denoted by *QID*, which is made of a unique peer-ID and a query counter managed by the query originator. Peers use *QID* to distinguish between new queries and those received before. After initialization, the query originator triggers the sequence of the following four phases: query forward, local query execution, merge-and-backward, and data retrieval. In all of these four phases, the communication between peers is done via APPA's Peer Communication service.

Query Forward
Q is included in a message that is broadcast to all reachable peers. Thus, like other flooding algorithms, each peer that receives *Q* tries to send it to its neighbors. Each peer *p* that receives the message including *Q* performs the following steps.
1. Check *QID*: if *Q* has been already received, then discard the message else save the address of the sender as the *parent* of *p*.
2. Decrement *TTL* by one: if *TTL* > *0*, make a new message including *Q*, *QID*, new *TTL* and the query originator's address and send the message to all neighbors (except parent).

In order to know their neighbors, the peers use the Peer Linking service of APPA.

Local Query Execution
After the query-forward phase, each peer *p* executes *Q* locally, *i.e.* accesses the local data items that match the query predicate, scores them using a scoring function, selects the *k* top data items and saves them as well as their scores locally. For scoring the data items, we can use one of the scoring functions proposed for relational data, *e.g.* Euclidean function [7][6]. These functions require no global information and can score peer's data items only using local information. The scoring function can also be specified explicitly by the user.

After selecting the *k* local top data items, *p* must wait to receive its neighbors' score-lists before starting the next phase. However, since some of the neighbors may leave the P2P system and never send a score-list to *p*, we must set a limit for the wait time. We compute *p*'s wait time using a cost function based on TTL, network dependent parameters and *p*'s local processing parameters. However, because of space limitations, we do not give the details of the cost function here.

Merge-and-Backward
After the wait time has expired, each peer merges its local top scores with those received from its neighbors and sends the result to its *parent* (the peer from which it received *Q*) in the form of a *score-list*. In order to minimize network traffic, we do not "bubble up" the top data items (which could be large), only their addresses. A score-list is simply a list of *k* couples *(p, s)*, such that *p* is the address of the peer owning the data item and *s* its score. Thus, each peer performs the following steps:
1. Merge the score-lists received from the neighbors with its local top scores and extracting the *k* top scores (along with the peer addresses).
2. Send the merged score-list to its parent.

Data Retrieval
After the query originator has produced the final score-list (gained by merging its local top scores with those received from its neighbors), it directly retrieves the *k* top data items from the peers in the list as follows. For each peer address *p* in the final score-list:
1. Determine the number of times *p* appears in the final score-list, *e.g. m* times.
2. Ask the peer at *p* to return its *m* top scored items.

Formally, consider the final score-list L_f which is a set of at most k couples *(p, s)*, in this phase for each $p \in Domain(L_f)$, the query originator determines $T_p = \{s \mid (p, s) \in L_f\}$ and asks peer p to return $\lceil T_p \rceil$ of its top scored items.

3.3 Analysis of Bandwidth Cost

One main concern with flooding algorithms is their bandwidth cost. In this section, we analyze our algorithm's bandwidth cost. As we will see, it is not very high. We also propose strategies to reduce it more. We measure the bandwidth cost in terms of number of messages and number of bytes which should be transferred over the network in order to execute a query by our algorithm. The messages transferred can be classified as: 1) *forward messages*, for forwarding the query to peers. 2) *backward messages*, for returning the score-lists from peers to the query originator. 3) *retrieve messages*, to request and retrieve the k top results. We first present a model representing the peers that collaborate on executing our algorithm, and then analyze the bandwidth cost of backward, retrieve and forward messages.

Model

Let P be the set of the peers in the P2P system. Given a query Q, let $P_Q \subseteq P$ be a set containing the query originator and all peers that receive Q. We model the peers in P_Q and the links between them by a graph $G(P_Q, E)$ where P_Q is the set of *vertices* in G and E is the set of the *edges*. There is an edge p-q in E if and only if there is a link between the peers p and q in the P2P system. Two peers are called *neighbor*, if and only if there is an edge between them in G. The number of neighbors of each peer $p \in P_Q$ is called the *degree of p* and is denoted by $d(p)$. The average degree of peers in G is called the *average degree of G* and is denoted by $d(G)$. The average degree of G can be computed as $d(G) = (\sum_{p \in P_Q} d(p)) / |P_Q|$

During the execution of our algorithm, $p \in P_Q$ may receive Q from some of its neighbors. The first peer, say q, which p receives Q from, is the *parent* of p in G, and thereby p is a *child* of q. A peer may have some neighbors that are neither its parent nor its children.

Backward Messages

In the Merge-and-Backward phase, each peer in P_Q, except the query originator, sends its merged score-list to its parent. Therefore, the number of backward messages, denoted by m_{bw}, is $m_{bw} = \lceil P_Q \rceil - 1$.

Let L be the size of each element of a score-list in bytes (*i.e.* the size of a score and an address), then the size of the score-list is $k \times L$, where k is the number of top results specified in Q. Since the number of score-lists transferred by backward messages is $\lceil P_Q \rceil - 1$, then the total size of data transferred by backward messages, denoted by b_{bw}, can be computed as $b_{bw} = k \times L \times (\lceil P_Q \rceil - 1)$. If we set $L=10$, *i.e.* 4 bytes for the score and 6 bytes for the address (4 bytes for IP address and 2 bytes for the port number), then $b_{bw} = k \times 10 \times (\lceil P_Q \rceil - 1)$.

Let us show with an example that b_{bw} is not significant. Consider that *10,000* peers receive Q (including the query originator), thus $/P_Q/=10{,}000$. Since users are interested in a few results and k is usually small, we set $k=20$. As a result, b_{bw} is less than 2 megabytes. Compared with the tens of megabytes of music and video files, which are typically downloaded in P2P systems, this is small.

Retrieve Messages

By retrieve messages, we mean the messages sent by the query originator to request the k top results and the messages sent by the peers owning the top results to return these results. In the Data Retrieval phase, the query originator sends at most k messages to the peers owning the top results (there may be peers owning more than one top result) for requesting their top results and these peers return their top results by at most k messages. Therefore, the number of retrieve messages, denoted by m_{rt}, is $m_{rt} \leq 2 \times k$.

Forward Messages

Forward messages are the messages that we use to forward Q to the peers. According to the basic design of our algorithm, each peer in P_Q sends Q to all its neighbors except its parent. Let p_o denote the query originator. Consider the graph $G(P_Q, E)$ described before, each $p \in (P_Q - \{p_o\})$, sends Q to $d(p)-1$ peers, where $d(p)$ is the degree of p in G. The query originator sends Q to all of its neighbors, in other words to $d(p_o)$ peers. Then, the sum of all forward messages m_{fw} can be computed as

$$m_{fw} = \left(\sum_{p \in (P_Q - \{p_o\})} (d(p) - 1) \right) + d(p_o)$$

We can write m_{fw} as follows:

$$m_{fw} = \left(\sum_{p \in P_Q} (d(p) - 1) \right) + 1 = \left(\sum_{p \in P_Q} (d(p)) \right) - |P_Q| + 1$$

Based on the definition of $d(G)$, m_{fw} can be written as $m_{fw} = (d(G) - 1) \times /P_Q/ + 1$, where $d(G)$ is the average degree of G. According to the measurements in [17], the average degree of Gnutella is 4. If we take this value as the average degree of the P2P system, *i.e.* $d(G)=4$, we have $m_{fw} = 3 \times /P_Q/ + 1$. From the above discussion, we can derive the following lemma.

Lemma 1: The number of forward messages in the basic form of our algorithm is $(d(G) - 1) \times /P_Q/ + 1$.

Proof: Implied by the above discussion. □

To determine the minimum number of messages necessary for forwarding Q, we prove the following lemma.

Lemma 2: The lower bound of the number of forward messages for sending Q to all peers in P_Q is $/P_Q/ - 1$.

Proof: For sending Q to each peer $p \in P_Q$, we need at least one forward message. Only one peer in P_Q has Q, *i.e.* the query originator, thus Q should be sent to $/P_Q/ - 1$ peers. Consequently, we need at least $/P_Q/ - 1$ forward messages to send Q to all peers in P_Q. □

Thus, the number of forward messages in the basic form of our algorithm is far from the lower bound.

3.4 Reducing the Number of Messages

We can still reduce the number of forward messages using the following strategies. 1) sending Q across each edge only once. 2) Sending with Q a list of peers that have received it.

Sending Q Across each Edge only once

In graph G, there may be many cases that two peers p and q are neighbors and none of them is the parent of the other, *e.g.* two neighbors which are children of the same parent. In these cases, in the basic form of our algorithm, both peers send Q to the other, *i.e.* Q is sent across the edge p-q twice. We develop the following strategy to send Q across an edge only once.

Strategy 1: When a peer p receives Q, say at time t, from its parent (which is the first time that p receives Q from), it waits for a random, small time, say λ, and then sends Q only to the neighbors which p has not received Q from them before $t + \lambda$.

Lemma 3: With a high probability, the number of forward messages with Strategy 1 is reduced to $d(G) \times |P_Q| / 2$.

Proof: Since λ is a random number and different peers generate independent random values for λ, the probability that two neighbors send Q to each other simultaneously is very low. Ignoring the cases where two neighbors send Q to the other simultaneously, with Strategy 1, Q is sent across an edge only once. Therefore, the number of forward messages can be computed as $m_{fw} = |E|$. Since $|E| = d(G) \times |P_Q| / 2$, then $m_{fw} = d(G) \times |P_Q| / 2$. □

Considering $d(G)=4$ (similar to [17]), the number of forward messages is $m_{fw} = 2 \times |P_Q|$.

With Strategy 1, m_{fw} is closer to the lower bound than the basic form of our algorithm. However, we are still far from the lower bound. By combining Strategy 1 and another strategy, we can reduce the number of forward messages much more.

Attaching to Forward Messages the List of Peers that have received Q

Even with Strategy 1, between two neighbors, which are children of the same parent p, one forward message is sent although it is useless (because both of them have received Q from p). If p attaches a list of its neighbors to Q, then its children can avoid sending Q to each other. Thus, we propose a second strategy.

Strategy 2: Before sending Q to its neighbors, a peer p attaches to Q a list containing its Id and the Id of its neighbors and sends this list along with Q. Each peer that receives the Q's message, verifies the list and does not send Q to the peers involved in the list.

Theorem 1: By combining Strategy 1 and Strategy 2, with a high probability, the number of forward messages is less than $d(G) \times |P_Q| / 2$.

Proof: With Strategy 2, two neighbors, which have the same parent, do not send any forward message to each other. If we use Strategy 1, with a high probability at most one forward message is sent across each edge. Using Strategy 2, there may be some edges such that no forward message is sent across them, *e.g.* edges between two neighbors with the same parent. Therefore, by combining Strategy 1 and Strategy 2, the number of forward messages is $m_{fw} \leq |E|$, and thus $m_{fw} \leq d(G) \times |P_Q| / 2$. □

Considering $d(G)=4$, the number of forward messages is $m_{fw} \leq 2 \times |P_Q|$.

4 Performance Evaluation

We evaluated the performance of our Fully Distributed algorithm (FD for short) through implementation and simulation. The implementation over a 64-node cluster was useful to validate our algorithm and calibrate our simulator. The simulation allows us to study scale up to high numbers of peers (up to 10,000 peers).

The rest of this section is organized as follows. In Section 4.1, we describe our experimental and simulation setup, and the algorithms used for comparison. In Section 4.2, we evaluate the response time of our algorithm. We first present experimental results using the implementation of our algorithm and four other baseline algorithms on a 64-node cluster, and then we present simulation results on the response time by increasing the number of peers up to *10,000*. We also did other experiments on the response time by varying other parameters, *e.g.* data item size, connection bandwidth, latency and k, but due to space limitation we cannot present them.

4.1 Experimental and Simulation Setup

For our implementation and simulation, we used the Java programming language, the SimJava package and the BRITE universal topology generator.

SimJava [11] is a process based discrete event simulation package for Java. Based on a discrete event simulation kernel, SimJava includes facilities for representing simulation objects as animated icons on screen. A SimJava simulation is a collection of entities each running in its own thread. These entities are connected together by ports and can communicate with each other by sending and receiving event objects.

BRITE [5] has recently emerged as one of the most promising universal topology generators. The objective of BRITE is to produce a general and powerful topology generation framework. Using BRITE, we generated topologies similar to those of P2P systems and we used them for determining the linkage between peers in our tests.

We first implemented our algorithm in Java on the largest set of machines that was directly available to us. The cluster has 64 nodes connected by a 1-Gbps network. Each node has an Intel Pentium 2.4 GHz processor, and runs the Linux operating system. We make each node act as a peer in the P2P system. To have a P2P topology close to real P2P overlay topologies, we determined the peer neighbors using the topologies generated by the BRITE universal topology generator [5]. Thus, each node only is allowed to communicate with the nodes that are its neighbors in the topology generated by BRITE.

To study the scalability of our algorithm far beyond 64 peers and to play with various performance parameters, we implemented a simulator using SimJava. To simulate a peer, we use a SimJava entity that performs all tasks that must be done by a peer for executing our algorithm. We assign a delay to communication ports to

simulate the delay for sending a message between two peers in a real P2P system. For determining the links between peers, we used the topologies generated by BRITE.

In all our tests, we use the following simple query as workload:

SELECT R.data FROM R ORDER BY R.score
STOP AFTER *k*

Each peer has a table *R(score, data)* in which attribute *score* is a random real number in the interval *[0..1]* with uniform distribution, and attribute *data* is a random variable with normal distribution with a mean of 1 (kilo bytes) and a variance of 64. Attribute *score* represents the score of data items and attribute *data* represents (the description of) the data item that will be returned back to the user as the result of query processing. The number of tuples of *R* at each peer is a random number (uniformly distributed over all peers) greater than 1000 and less than 20,000.

The simulation parameters are shown in Table 1. Unless otherwise specified, the latency between any two peers is a normally distributed random number with a mean of 200 (ms) and a variance of 100. The bandwidth between peers is also a random number with normal distribution with a mean of 56 (kbps) and a variance of 32. Since users are usually interested in a small number of top results, we set *k=20*.

The simulator allows us to perform tests up to *10,000* peers, after which the simulation data no longer fit in RAM and makes our tests difficult. This is quite sufficient for our tests. Therefore, the number of peers of P2P system is set to be *10,000*, unless otherwise specified. In all tests, *TTL* is set as the maximum hop-distance to other peers from the query originator, thus all peers of the P2P system can receive *Q*. We observed that in the topologies with *10,000* nodes, with *TTL=12* all peers could receive *Q*. Our observations correspond to those based on experiments with the Gnutella network [17]; for instance, with *50,000* nodes, the maximum hop-distance between any two nodes is *14*.

Table 1. Simulation parameters

Parameter	Values
Bandwidth	Normally distributed random, Mean = 56 Kbps, Variance = 32
Latency	Normally distributed random, Mean = 200 ms, Variance = 100
Number of peers	10,000 peers
TTL	Large enough such that all of peers can receive the query
K	20
Result items size	Normally distributed random, Mean = 1 KB, Variance = 64

In our simulation, we compare our FD algorithm with four other algorithms. The first algorithm is a *Naïve* algorithm that works as follows. Each peer receiving *Q* sends its *k* top relevant items directly to the query originator. The query originator merges the received results and extracts the *k* overall top scored data items from them.

The second algorithm is an adaptation of Edutella's algorithm [21] which is designed for super-peer. We adapt this algorithm for an unstructured system and call it *Sequential Merging (SM)* as it sequentially merges top data items. The original Edutella algorithm works as follows. The query originator sends *Q* to its super-peer, and it sends *Q* to all other super-peers. The super-peers send *Q* to the peers connected to them. Each peer that has data items relevant to *Q* scores them and sends its

maximum scored data item to its super-peer. Each super-peer chooses the overall maximum scored item from all received data items. For determining the second best item, it only asks one peer, the one which returned the first top item, to return its second top scored item. Then, the super-peer selects the overall second top item from the previously received items and the newly received item. Then, it asks the peer which returned the second top item and so on until all k top items will be retrieved. Finally the super-peers send their top items to the super-peer of the query originator, to extract overall k top items, and to send them to the query originator. In Edutella, a very small percentage of nodes are super-peers, *e.g.* in [19] it is 0.64, *i.e.* 64 super-peers for 10,000 peers. In our tests, we consider the same percentage, and we select the super-peers randomly from the peers of P2P system. We consider the same computing capacity for the super-peers as for the other peers.

We also propose the optimized versions of Naïve and SM algorithms that bubble up only the score-lists, as in our algorithm, and we denote them *Naïve** and *SM** respectively. In our tests, in addition to Naïve and SM algorithms, we compare our algorithm with Naïve* and SM*.

4.2 Scale up

In this section, we investigate the scalability of our algorithm. We use both our implementation and simulator to study response time while varying the number of peers. The response time includes local processing time and data transfers, *i.e.* sending query messages, score-lists and data items.

Using our implementation over the cluster, we ran experiments to study how response time increases with the addition of peers. Figure 2 shows excellent scale up of our algorithm since response time logarithmically increases with the addition of peers until 64. Using simulation, Figure 3 shows the response times of the five algorithms with a number of peers increasing up to 10000 and the other simulation parameters set as in Table 1.

FD always outperforms the four other algorithms and the performance difference increases significantly in favor of FD as the number of peers increases. The main reason for FD's excellent scalability is its fully distributed execution. With the SM, SM*, Naive and Naïve*, a central node is responsible for query execution, and this creates two problems. First, the central node becomes a communication bottleneck since it must receive a large amount of data from other peers that all compete for bandwidth. Second, the central node becomes a processing bottleneck, as it must merge many answers to extract the k top results.

Another advantage of FD is that it does not transfer useless data items over the network. For determining top items, FD only bubbles up the score-lists (which are small) while SM and Naive algorithms transfer many data items of which only a small fraction makes the final top results. SM transfers the first top-scored item of every peer and Naïve transfers k top-scored data items of all peers. With a large number of peers, data transfer is a dominant factor in response time and FD reduces it to minimum.

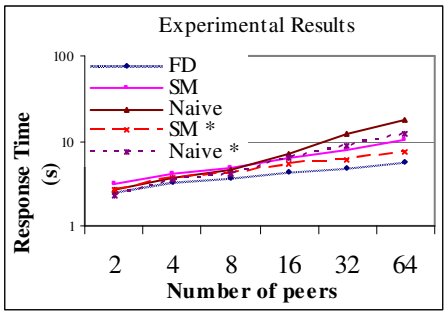

Fig. 2. *Response time vs. number of peers*

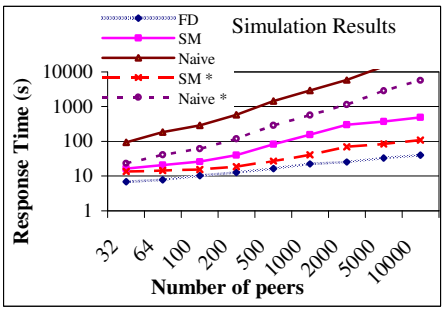

Fig. 3. *Response time vs. number of peers*

Overall, the experimental results correspond with the simulation results. However, the response time gained from our experiments over the cluster is a little better than that of simulation because the cluster has a high-speed network.

We also did experiments on the response time by varying data item size, connection bandwidth, latency and k. The item size has little impact on the response time of FD, SM* and Naïve*, but has strong impact on SM and Naïve. The response time decreases with increasing the connection bandwidth in all five algorithms. However, FD outperforms the other algorithms for all the tested bandwidths. FD also outperforms the other algorithms for the tested values of latency (up to 10,000 ms). However, high latency, *e.g.* more than 2000 ms, has strong impact and increases response time much, but below 2000 ms, latency has not much effect on FD's response time. According to studies reported in [18], more than 80% of links between peers have good latency, less than 280 ms, for which FD has very good performance. k has little impact on the response time of SM, but has some impact on FD, SM*, Naïve and Naïve*. Despite the effect of k on FD, it is by far the superior algorithm for the tested values of k ($k<200$). Since users are usually interested in a small number of top results, *e.g.* less than *20* results, the performance advantage for FD remains high.

5 Related Work

In the context of P2P systems, little research has concentrated on Top-k query processing. In [21] the authors present a Top-k query processing algorithm for Edutella which is a super-peer network. The technique which Edutella uses for processing Top-k queries is explained in Section 4.1. Although very good for super-peer networks, this technique cannot apply efficiently to other networks, in particular, unstructured, since there may be no peer with higher reliability and computing power. In contrast, our algorithm makes no assumptions about the P2P network topology and the existence of certain peers.

A good formal framework for ranking is introduced in [1] based on a ranking algebra. The authors show that not only one global ranking should be taken into account, but also several in different contexts. The ranking algebra allows aggregating the local rankings into global rankings.

PlanetP [8] is a P2P system that constructs a content addressable publish/subscribe service using gossiping to replicate global documents across P2P communities up to ten thousand peers. In PlanetP, a Top-k query processing method is proposed that works as follows. Given a query Q , the query originator computes a relevance ranking of peers with respect to Q, contacts them one by one from top to bottom of ranking and asks them to return a set of their top-scored document names together with their scores. To compute the relevance of peers, a global fully replicated index is used that contains term-to-peer mappings. In a large P2P system, keeping up-to-date the replicated index is a major problem that hurts scalability. In contrast, our algorithm does not use any replicated data.

For the cases where a data item can have multiple scores at different sites, *e.g.* the amount of a customer's purchase in several stores, the TA family of algorithms for monotonic score aggregation [9] stands out as an efficient and highly versatile method. There have been many algorithms in order to optimize the TA algorithm in terms of bandwidth cost and response time, *e.g.* [22] and [14].

6 Conclusion

In this paper, we proposed a fully distributed algorithm for Top-k query processing in the context of the APPA data management system. APPA has a network-independent design that can be implemented over different P2P networks (unstructured, DHT, super-peer, etc.), thus allowing us to exploit continuing progress in such systems. We presented our algorithm for the case of unstructured systems, thus with minimal assumptions. Our algorithm requires no global information, does not depend on the existence of certain peers and its bandwidth cost is low.

For determining the k top results, we use the concept of score-list which reduces the bandwidth consumption and also reduces the response time. We analyzed the bandwidth cost of our algorithm and we proposed efficient techniques in order to reduce it.

We validated the performance of our algorithm through implementation over a 64-node cluster and simulation using the BRITE topology generator and SimJava. The experimental and simulation results showed that our algorithm can have logarithmic scale up. The simulation also showed the excellent performance of our algorithm compared with a naïve algorithm and an adaptation of an existing algorithm.

As future work, we plan to deal with replicated data in P2P Top-k query processing. In this paper, we assumed that data items are not replicated. In the case of data replication, with our algorithm, there may be replicated data items in the final score-list. This may be fine for the user as it is an indication of the items' usefulness (in a P2P system, the most useful data get most replicated). But we could also identify replicated items.

References

[1] Aberer, K., AND Wu., J. Framework for Decentralized Ranking in Web Information Retrieval. *Proc. of the 5th Asia Pacific Web Conference (APWeb)*, 2003.

[2] Abiteboul, S., et al. Dynamic XML documents with distribution and replication. *SIGMOD Conf.*, 2003.

[3] Akbarinia, R., Martins, V., Pacitti, E., and Valduriez, P. Design and Implementation of Atlas P2P Architecture. *Global Data Management* (Eds. R. Baldoni, G. Cortese, F. Davide), IOS Press, 2006.

[4] Akbarinia, R., Martins, V., Pacitti, E., AND Valduriez, P. Replication and Query Processing in the APPA Data Management System. *6th Workshop on Distributed Data & Structures (WDAS)*, 2004.

[5] BRITE, http://www.cs.bu.edu/brite/.

[6] Carey, M.J., AND Kossmann, D. On saying 'Enough Already!'. *SIGMOD Conf.*, 1997.

[7] Chaudhuri, S., et al. Evaluating Top-k Selection queries. *VLDB Conf.*, 1999.

[8] Cuenca-Acuna, F.M., Peery, C., Martin, R.P., AND Nguyen, T.D. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. *IEEE Int. Symp. on High Performance Distributed Computing (HPDC)*, 2003.

[9] Fagin, R., Lotem, J., AND Naor, M. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci. 66(4)*, 2003.

[10] Gnutella. http://www.gnutelliums.com/.

[11] Howell, F., AND McNab, R. SimJava: a discrete event simulation package for Java with applications in computer systems modeling. *Int. Conf. on Web-based Modelling and Simulation, San Diego CA, Society for Computer Simulation*, 1998.

[12] Huebsch, R., et al. Querying the Internet with PIER. *VLDB Conf.*, 2003.

[13] Kazaa. http://www.kazaa.com/.

[14] Michel, S., Triantafillou, P., AND Weikum, G. KLEE: A Framework for Distributed Top-k Query Algorithms. *VLDB Conf.*, 2005.

[15] Ooi, B., Shu, Y., AND Tan, K-L. Relational data sharing in peer-based data management systems. *SIGMOD Record, 32(3),* 2003.

[16] Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., AND Shenker, S. A scalable content-addressable network. *Proc. of SIGCOMM*, 2001.

[17] Ripeanu, M., AND Foster, I. Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. *IPTPS*, 2002.

[18] Saroiu, S., Gummadi, P., AND Gribble, S. A Measurement Study of Peer-to-Peer File Sharing Systems. *Proc. of Multimedia Computing and Networking (MMCN)*, 2002.

[19] Siberski, W., AND Thaden, U. A Simulation Framework for Schema-Based Query Routing in P2P-Networks. *EDBT Workshops*, 2004.

[20] Tatarinov, I., et al. The Piazza peer data management project. *SIGMOD Record 32(3)*, 2003.

[21] Thaden, U., Siberski, W., Balke, W.T., AND Nejdl, W. Top-k query Evaluation for Schema-Based Peer-To-Peer Networks, *Int. Semantic Web Conf. (ISWC)*, 2004.

[22] Theobald, M., Weikum, G., AND Schenkel, R. Top-k Query Evaluation with Probabilistic Guarantees. *VLDB Conf.*, 2004.

[23] Yang, B., AND Garcia-Molina, H. Designing a super-peer network. *Int. Conf. on Data Engineering*, 2003.

[24] Yu, C., et al. Databases Selection for Processing k Nearest Neighbors Queries in Distributed Environments. *ACM/IEEE-CS joint Conf. on DL*, 2001.

[25] Yu, C., Philip, G., AND Meng, W. Distributed Top-n Query Processing with Possibly Uncooperative Local Systems, *VLDB Conf.*, 2003.