As-Soon-As-Possible Top-k Query Processing in P2P Systems

William Kokou Dedzoe[1], Philippe Lamarre[2], Reza Akbarinia[3], and Patrick Valduriez[3]

[1] INRIA Rennes, France
William.Dedzoe@inria.fr
[2] INSA de Lyon, France
Philippe.Lamarre@insa-lyon.fr
[3] INRIA and LIRMM, Montpellier, France
{Reza.Akbarinia, Patrick.Valduriez}@inria.fr

Abstract. Top-k query processing techniques provide two main advantages for unstructured peer-to-peer (P2P) systems. First they avoid overwhelming users with too many results. Second they reduce significantly network resources consumption. However, existing approaches suffer from long waiting times. This is because top-k results are returned only when all queried peers have finished processing the query. As a result, query response time is dominated by the slowest queried peer. In this paper, we address this users' waiting time problem. For this, we revisit top-k query processing in P2P systems by introducing two novel notions in addition to response time: the *stabilization time* and the *cumulative quality gap*. Using these notions, we formally define the as-soon-as-possible (ASAP) top-k processing problem. Then, we propose a family of algorithms called ASAP to deal with this problem. We validate our solution through implementation and extensive experimentation. The results show that ASAP significantly outperforms baseline algorithms by returning final top-k result to users in much better times.

1 Introduction

Unstructured *Peer-to-Peer* (P2P) systems have gained great popularity in recent years and have been used by millions of users for sharing resources and content over the Internet [4, 30, 25]. In these systems, there is neither a centralized directory nor any control over the network topology or resource placement. Because of few topological constraints, they require little maintenance in highly dynamic environnements [26]. However, executing queries over unstructured P2P systems typically by flooding may incur high network traffic and produce lots of query results.

To reduce network traffic and avoid overwhelming users with high numbers of query results, complex query processing techniques based on top-k answers have been proposed e.g. in [2]. With a top-k query, the user specifies a number k of the most relevant answers to be returned by the system. The quality (i.e. score of relevance) of the answers to the query is determined by user-specified

scoring functions [9, 18]. Despite the fact that these top-k query processing solutions e.g. [2] reduce network traffic, they may significantly delay the answers to users. This is because top-k results are returned to the user only when all queried peers have finished processing the query. Thus, query response time is dominated by the slowest queried peer, which makes users suffer from long waiting times. Therefore, these solutions are not suitable for emerging applications such as P2P data sharing for online communities, which may have high numbers of autonomous data sources with various access performance. Most of the previous work on top-k processing has focused on efficiently computing the exact or approximate result sets and reducing network traffic [6, 17, 34, 32, 2].

A naive solution to reduce users' waiting time is to have each peer return its top-k results directly to the query originator as soon as it has done executing the query. However, this significantly increases network traffic and may cause a bottleneck at the query originator when returning high numbers of results. In this paper, we aim at reducing users' waiting time by returning high quality intermediate results, while avoiding high network traffic. The intermediate results are the results of peers which have already processed locally their query. Providing intermediate results to users is quite challenging because a naive solution may saturate users with results of low quality, and incur significant network traffic which in turn may increase query response time.

In this paper, our objective is to return high quality results to users as soon as possible. For this, we revisit top-k query processing in P2P systems by introducing two notions to complement response time: *stabilization time* and *cumulative quality gap*. The stabilization time is the time needed to obtain the final top-k result set, which may be much lower than the response time (when it is sure that there is no other top-k result). The quality gap of the top-k intermediate result set is the quality that remains to be the final top-k result set. The cumulative quality gap is the sum of the quality gaps of all top-k intermediate result sets during query execution. Using these notions, we formally define the as-soon-as-possible (ASAP) top-k processing problem. Then, we propose a family of algorithms called ASAP to deal with this problem.

This paper is an extended version of [12] with the following added value. First, in Section 6 we propose a solution to deal with node failures (or departures) which may decrease the quality and accuracy of top-k results. In Section 7, we propose two techniques to compute "probabilistic guarantees" for the users showing for example the probability that current intermediate top-k results are the true top-k results (i.e. confidence of current top-k result). Section 8.2 shows experimentally the effectiveness of our solution for computing "probabilistic guarantees". Finally, we study experimentally the impact of data distribution on our algorithms (Section 8.2).

2 System Model

In this section, we first present a general model of unstructured P2P systems which is needed for describing our solution. Then, we provide a model and definitions for top-k queries.

2.1 Unstructured P2P Model

We model an unstructured P2P network of n peers as an undirected graph $G = (P, E)$, where $P = \{p_0, p_1, \cdots, p_{n-1}\}$ is the set of peers and E the set of connections between the peers. For $p_i, p_j \in P, (p_i, p_j) \in E$ denotes that p_i and p_j are neighbours. We also denote by $N(p_i)$, the set of peers to which p_i is directly connected, so $N(p_i) = \{p_j | (p_i, p_j) \in E\}$. The value $\|N(p_i)\|$ is called the degree of p_i. The average degree of peers in G is called the *average degree* of G and is denoted by φ. The *r-neighborhood* $N^r(p)$ $(r \in \mathbb{N})$ of a peer $p \in P$ is defined as the set of peers which are at most r hops away from peer p, so

$$N^r(p) = \begin{cases} \{p\} & \text{if } r = 0 \\ \{p\} \bigcup\limits_{p' \in N(p)} N^{r-1}(p') & \text{if } r \geq 1 \end{cases}$$

Each peer $p \in P$ holds and maintains a set $D(p)$ of data items such as images, documents or relational data (i.e. tuples). We denote by $D^r(p)(r \in \mathbb{N})$, the set of all data items which are in $N^r(p)$, so

$$D^r(p) = \bigcup_{p' \in N^r(p)} D(p')$$

In our model, the query is forwarded from the query originator to its neighbours until the Time-To-Live value of the query decreases to 0 or the current peer has no peer to forward the query. So the query processing flow can be represented as a tree, which is called the query forwarding tree. When a peer $p_0 \in P$ issues query q to peers in its *r-neighborhood*, the results of these peers are bubbled up using query q's forwarding tree with root p_0 including all the peers belonging to $N^r(p_0)$. The set of children of a peer $p \in N^r(p_0)$ in query q's forwarding tree is denoted by $\psi(p, q)$.

2.2 Top-k Queries

We characterize each top-k query q by a tuple $< qid, c, ttl, k, f, p_0 >$ such that qid is the query identifier, c is the query itself (e.g. SQL query), $ttl \in \mathbb{N}$ (Time-To-Live) is the maximum hop distance set by the user, $k \in \mathbb{N}^*$ is the number of results requested by the user $f : \mathcal{D} \times \mathcal{Q} \rightarrow [0,1]$ is a scoring function that denotes the score of relevance (i.e. the quality) of a given data item with respect to a given query and $p_0 \in P$ the originator of query q, where $\mathcal{D}$ is the set of data items and $\mathcal{Q}$ the set of queries.

A top-k result set of a given query q is the k top results among data items owned by all peers that receive q. Formally we define this as follows.

Definition 1 ***Top-k Result Set****. Given a top-k query q, let $D' = D^{q.ttl}(q.p_0)$. The top-k result set of q, denoted by $Top^k(D', q)$, is a sorted set on the score (in decreasing order) such that:*

1. $Top^k(D', q) \subseteq D'$;
2. *If* $\|D'\| < q.k$, $Top^k(D', q) = D'$, *otherwise* $\|Top^k(D', q)\| = q.k$;
3. $\forall d \in Top^k(D', q)$, $\forall d' \in D' \setminus Top^k(D', q)$, $q.f(d, q.c) \geq q.f(d', q.c)$

Definition 2 ***Result's Rank****. Given a top-k Result set I, we define the rank of result $d \in I$, denoted by $rank(d, I)$, as the position of d in the set I.*

Note that the rank of a given top-k item is in the interval $\left[1; k\right]$.

In large unstructured P2P systems, peers have different processing capabilities and store different volumes of data. In addition, peers are autonomous in allocating the resources to process a given query. Thus, some peers may process more quickly a given query than others. Intuitively, the top-k intermediate result set for a given peer is the k best results of both the results the peer received so far from its children and its local results (if any). Formally, we define this as follows.

Definition 3 ***Top-k Intermediate Result Set****. Given a top-k query q, and $p \in N^{q.ttl}(q.p_0)$. Let D_1 be the result set of q received so far by p from peers in $\psi(p, q)$ and $D_2 = D_1 \cup D(p)$. The top-k intermediate result set of q at peer p, denoted by $I_q(p)$, is such that:*

$$I_q(p) = \begin{cases} Top^k(D_2, q) \text{ if } p \text{ has already processed } q \\ \\ Top^k(D_1, q) \text{ otherwise} \end{cases}$$

3 Problem Definition

Let us first give our assumptions regarding schema management and the unstructured P2P architecture. We assume that peers are able to express queries over their own schema without relying on a centralized global schema as in data integration systems [28]. Several solutions have been proposed to support decentralized schema mapping. However, this issue is out of scope of this paper and we assume it is provided using one of the existing techniques, *e.g.* [23], [28] and [1]. We also assume that all peers in the system are trusted and cooperative. In the following, we first give some definitions which are useful to define the problem we focus and formally state the problem.

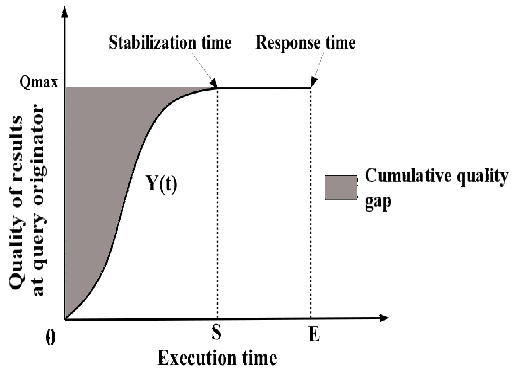


Fig. 1. Quality of top-k results at the query originator wrt. Execution time

3.1 Foundations

To process a top-k query in P2P systems, an ASAP top-k algorithm provides intermediate results to users as soon as peers process the query locally. This allows users to progressively see the evolution of their query execution by receiving intermediate results for their queries. Note that at some point of query execution, the top-k intermediate results received by a peer may not change any more, until the end of the query execution. We denote this point as the **stabilization time** (see Figure 1).

Recall that the main goal of ASAP top-k query processing is to return high-quality results to user as soon as possible. To reflect this, we introduce the *quality evolution* concept. Given a top-k query q, we define the quality evolution $Y(t)$ of q at time t as the sum of scores of q's intermediate top-k results at t and at q's originator. Figure 1 shows the quality evolution of intermediate top-k results obtained at the query originator during a given query execution. To be independent of the scoring values —which can be different from one query to another—, we normalize the quality evolution of a query. With this in mind, we divide the quality evolution of a given query by the sum of scores of the final top-k results of that query. Thus, the quality evolution values are in the interval $[0, 1]$ and the quality of the top-k final results is equal to 1. Note that we do not use the proportion of the final top-k results in intermediate top-k results (i.e precision) to characterize ASAP algorithm because this metric does not express the fact of returning the high quality results as soon as possible to users.

The quality evolution of intermediate top-k results at the query originator increases as peers answer the query. To reflect this, we introduce the *cumulative quality gap*, which is defined as the sum of the quality difference between intermediate top-k result sets received until the stabilization time and the final top-k result set. We formalize this in Definition 4.

Definition 4 ***Cumulative quality gap***. *Given a top-k query q, let $Y(t)$ be the quality evolution of q at time t at q originator, and S be the stabilization time*

of q. The cumulative quality gap of the query q, denoted by C_{qg} is:

$$C_{qg} = \int_0^S (1 - Y(t))\,\mathrm{d}t = S - \int_0^S Y(t)\,\mathrm{d}t \tag{1}$$

3.2 Problem Statement

Formally, we define the ASAP top-k query processing problem as follows. Given a top-k query q, let S be the stabilization time of q and C_{qg} be the cumulative quality gap of q. The problem is to minimize C_{qg} and S while avoiding high communication cost.

4 ASAP Top-k Query Processing Overview

ASAP query processing proceeds in two main phases. The first phase is the query forwarding and local execution of the query. The second phase is the bubbling up of the peers' results for the query along the query forwarding tree.

4.1 Query Forwarding and Local Execution

Query processing starts at the query originator, i.e. the peer at which a user issues a top-k query q. The query originator performs some initialization. First, it sets *ttl* which is either user-specified (or default). Second, it creates a unique identifier *qid* for q which is useful to distinguish between new queries and those received before. Then, q is included in a message that is broadcast by the query originator to its reachable neighbors. **Algorithm 1** shows the pseudo-code of query forwarding. Each peer that receives the message including q checks *qid* (see line 2, Algorithm 1). If it is the first time the peer has received q, it saves the query (i.e. saves the query in the list of seen queries and the address of the sender as its parent) and decreases the query *ttl* by 1 (see lines 3-4, Algorithm 1). If the *ttl* is greater than 0, then the peer sends the query message to all neighbors except its parent (see lines 5-7, Algorithm 1). Then, it executes q locally. If q has been already received, then if the old *ttl* is smaller than the new *ttl*, the peer proceeds as where q is received for the first time but without executing q locally (see lines 10-18, Algorithm 1), else the peer sends a duplicate message to the peer from which it has received q.

4.2 Bubbling Up Results

Recall that, when a peer submits a top-k query q, the local results of the peers who have received q are bubbled (i.e returned upwards) up to the query originator using query q's forwarding tree. In ASAP, a peer's decision to send intermediate results to its parent is based on the improvement impact computed by using

Algorithm 1: *receive_Query(msg)*

```
    input  : msg, a query message.
 1  begin
 2      if (!already_Received(msg.getID()) then
 3          memorize(msg);
 4          msg.decreaseTTL();
 5          if (msg.getTTL() > 0) then
 6              forwardToNeighbors(msg);
 7          end
 8          executeLocally(msg.getQuery());
 9      else
10          qid = msg.getID();
11          oldMsg = SeenQuery(qid).;
12          if (msg.getTTL() > oldMsg.TTL()) then
13              memorize(msg);
14              msg.decreaseTTL();
15              if (msg.getTTL() > 0) then
16                  forwardToNeighbors(msg);
17              end
18              sendDuplicateSignal(qid, oldMsg.getSender());
19          else
20              sendDuplicateSignal(qid, msg.getSender());
21          end
22      end
23  end
```

the ratio of its current top-k intermediate result set over the top-k intermediate result set which it has sent so far to its parent. This improvement impact can be computed in two ways: by using the score or rank of top-k results in the result set. Therefore, we introduce two types of improvement impact: *score-based improvement impact* and *rank-based improvement impact*.

Intuitively, the score-based improvement impact at a given peer for a given top-k query is the gain of score of that peer's current top-k intermediate set compared to the top-k intermediate set it has sent so far.

Definition 5 ***Score-based improvement impact****. Given a top-k query q, and peer $p \in N^{q.ttl}(q.p_0)$, let T_{cur} be the current top-k intermediate set of q at p and T_{old} be the top-k intermediate set of q sent so far by p. The score-based improvement impact of q at peer p, denoted by $IScore(T_{cur}, T_{old})$ is computed as*

$$IScore(T_{cur}, T_{old}) = \frac{\sum_{d \in T_{cur}} q.f(d, q.c) - \sum_{d' \in T_{old}} q.f(d', q.c)}{k} \quad (2)$$

Note that in Formula 2, we divide by k instead of $\|T_{cur} - T_{old}\|$ because we do not want that $IScore(T_{cur}, T_{old})$ be an average which would not be very sensitive to the values of scores. The score-based improvement impact values are in the interval $[0, 1]$.

Intuitively, the rank-based improvement impact at a given peer for a given top-k query is the loss of rank of results in the top-k intermediate result set sent so far by that peer due to the arrival of new intermediate results.

Definition 6 ***Rank-based improvement impact***. *Given a top-k query q and peer $p \in N^{q.ttl}(q.p_0)$, let T_{cur} be the current top-k intermediate result set of q at p and T_{old} be the top-k intermediate result set of q sent so far by p. The rank-based improvement impact of q at peer p, denoted by $IRank(T_{cur}, T_{old})$ is computed as*

$$IRank(T_{cur}, T_{old}) = \frac{\sum_{d \in T_{cur} \setminus T_{old}} (k - rank(d, T_{cur}) + 1)}{\frac{k * (k+1)}{2}} \quad (3)$$

Note that in Formula 3, we divide by $\frac{k*(k+1)}{2}$ which is the sum of ranks of a set containing k items. The rank-based improvement impact values are in the interval $[0, 1]$.

Notice also that, in order to minimize network traffic, ASAP does not bubble up the results (which could be large), but only their scores and addresses. A score-list is simply a list of k pairs (ad, s), such that ad is the address of the peer owning the data item and s its score.

A simple way to decide when peer must bubble up newly received intermediate results to its parent is to set a minimum value (threshold) that must reach its improvement impact. This value is set initially by the application and it is the same for all peers in the system. Note also that this threshold does not change during the execution of the query. Using both types of improvement impact we have introduced, we have two types of static threshold-based approaches. The first approach uses the score-based improvement impact and the second one the rank-based improvement impact.

A generic algorithm for our static threshold-based approaches is given in **Algorithm 2**. In these approaches, each peer maintains for each query a set T_{old} of top-k intermediate results sent so far to its parent and a set T_{cur} of current top-k intermediate results. When a peer receives a new result set N from its children (or its own result set after local processing of a query), it first updates the set T_{cur} with results in N (see line 2, Algorithm 2). Then, it computes the improvement impact *imp* of T_{cur} compared to T_{old} (line 3, Algorithm 2). If *imp* is greater than or equal to the defined threshold *delta* or if there are no more children' results to wait for, the peer sends the set $T_{tosend} = T_{cur} \setminus T_{old}$ to its parent and subsequently sets T_{curr} to T_{old} (see lines 4-7, Algorithm 2).

5 Dynamic Threshold-based Approaches for Bubbling up Results

Although the static threshold-based approaches are interesting to provide results quickly to user, they may be blocking if results having higher scores are bubbled up before those of lower score. In other words, sending higher score's results will induce a decrease of improvement impact of the following results. This is because the improvement impact considers the top-k intermediate results sent so far by

Algorithm 2: $Streat(k, T_{cur}, T_{old}, N, delta, Func)$

```
input : k, number of results; T_cur, current top-k; T_old, top-k sent so far; N, new
        result set; delta, impact threshold; Func, type of improvement impact.
1 begin
2   T_cur = mergingSort_Topk(k, T_cur, N);
3   imp = Func(T_cur, T_old);
4   if ((imp ≥ delta) or all_Results()) then
5     T_tosend = T_cur \ T_old;
6     send_Parent(T_tosend, all_Results());
7     T_old = T_cur;
8   end
9 end
```

the peer. Thus, results of low scores even if they are in the final top-k results may be returned at the end of the query execution. To deal with this problem, an interesting way would be to have a dynamic threshold, i.e. a threshold that decreases as the query execution progresses. However, this would require finding the right parameter on which the threshold depends. We have identified two possible solutions for the dynamic threshold. The first one is to use an estimation of the query execution time. However, estimating the query execution time in large P2P system is very difficult because it depends on network dynamics, such as connectivity, density, medium access contention, etc., and the slowest queried peer. The second, more practical, solution is to use the peer's result set coverage, i.e for each peer the proportion of peers in its sub-tree including itself (i.e. all its descendants and itself) which have already processed the query to decrease the threshold.

5.1 Peer's Local Result Set Coverage

Definition 7 ***Peer's local result set coverage****. Given a top-k query, and* $p \in N^{q.ttl}(q.p_0)$*, let* $\mathcal{A}$ *be the set of peers in the sub-tree whose root is p in the query q's forwarding tree. Let* $\mathcal{E}$ *be the set of peers in* $\mathcal{A}$ *which have already processed q locally. The local result set coverage of peer p for q, denoted by* $Cov(\mathcal{E}, \mathcal{A})$*, is computed using the following equation:*

$$Cov(\mathcal{E}, \mathcal{A}) = \frac{\|\mathcal{E}\|}{\|\mathcal{A}\|}$$

Peer's local result set coverage values are in the interval $[0, 1]$.

Note that is very difficult to have the exact value of a peer's local result set coverage without inducing an additional number of messages in the network. This is because each peer must send a message to its parent each time its local coverage result set value changes. Thus, when a peer at hop m from query originator updates its local result coverage, m messages will be sent over the network. To deal with this problem, an interesting solution is to have an estimation of this value instead of the exact value.

The estimation of peer's local result set coverage can be done using two different strategies: optimistic and pessimistic. In the optimistic strategy, each

peer computes the initial value of its local result set coverage based only on its children nodes. This value is then updated progressively as the peers in its sub-tree bubble up their results. Indeed, each peer includes in each response message sent to its parent the number of peers in its sub-tree (including itself) which have already processed the query locally and the total number of peers in its sub-tree including itself. This couple of values is used in turn by its parent to estimate its local result set coverage. Contrary to the optimistic strategy, in the pessimistic strategy, the local result set coverage estimation is computed at the beginning by each peer based on the Time-To-Live received with the query and the average degree of peers in the system. As in the case of the optimistic strategy, this value is updated progressively as the peers in its sub-tree bubble up their results.

In our dynamic threshold-based approaches, we estimate a peer's local result set coverage using the pessimistic strategy because the estimation value is more stable than with the optimistic strategy. Now, let us give more details about how a peer's local result set coverage pessimistic estimation strategy is done.

5.2 Peer's Local Result set Coverage Pessimistic Estimation

In order to estimate its local result set coverage, each peer p_i maintains for each top-k query q and for each child p_j a set $\mathcal{C}_1$ of pairs (p_j, a) where $a \in \mathbb{N}$ is the number of peers in the sub-tree of peer p_j including p_j itself. p_i maintains also a set $\mathcal{C}_2$ of pairs (p_j, e) where $e \in \mathbb{N}$ is the total number of peers in the sub-tree of peer p_j including p_j itself which have already processed locally q. Now let ttl' be the time-to-live with which p_i received query q and φ be the average degree of peers in the system. At the beginning of query processing, for all children of p_i, $e = 0$ and $a = \sum_{u=0}^{ttl'-2} \varphi^u$. During query processing, when a child p_j in $\psi(p_i, q)$ wants to send results to p_i, it inserts in the answer message its couple of values (e, a). Once p_i receives this message, it unpacks the message, gets these values (i.e. e and a) and updates the sets $\mathcal{C}_1$ and $\mathcal{C}_2$. The local result set coverage of peer p_i for the query q is then estimated using Formula 4.

$$\widetilde{Cov}(\mathcal{C}_1, \mathcal{C}_2) = \frac{\sum_{(p_j, e) \in \mathcal{C}_1} e}{\sum_{(p_j, a) \in \mathcal{C}_2} a} \tag{4}$$

Note that peer's local result set coverage estimation values are in the interval $[0, 1]$.

5.3 Dynamic Threshold Function

In the dynamic threshold approaches, the improvement impact threshold used by a peer at a given time t of the query execution depends on its local result set

coverage at that time. This improvement impact threshold decreases as the local result set coverage increases. To decrease the improvement impact threshold used by a peer as the local result set coverage increases, we use a linear function that allows peers to set their improvement impact threshold for a given local result set coverage. Now let us define formally the threshold function.

Definition 8 ***Dynamic Threshold Function****. Given a top-k query q and $p \in N^{q.ttl}(q.p_0)$, the improvement impact threshold used by p during q's execution, is a monotonically decreasing function H such that:*

$$H : \begin{vmatrix} [0,1] \rightarrow [0,1] \\ \\ x \quad \mapsto -\alpha * x + \alpha \end{vmatrix} \tag{5}$$

with $\alpha \in [0, 1[$. Notice that x is a peer's result set coverage at given time and α the initial improvement impact threshold (i.e. $H(0) = \alpha$).

5.4 Reducing Communication Cost

Using a rank-based improvement impact has the drawback of not reducing as much as possible network traffic. This is because the rank-based improvement impact value is equal to 1 (the maximum value it can reach) when a peer receives the first result set (from one of its children or after local processing of a query). Thus, each peer always sends a message over the network when it receives the first result set containing k results. To deal with this problem and thus reduce communication cost, we use peers' result sets coverage to prevent them to send a message when they receive their first result set. Therefore, the idea is to allow peers to start sending a message if and only if their local result sets coverage reaches a predefined threshold. With this result set coverage threshold, peers send intermediate results based on the improvement impact threshold obtained from the dynamic threshold function H define above.

5.5 Dynamic Threshold Algorithms

Our dynamic threshold approaches algorithms are based on the same principles as the static threshold ones. A generic algorithm for our dynamic threshold-based approaches is given in **Algorithm 3**. When a peer receives a new result set N from its children (or generates its own result set after local processing of a query), it first updates the set T_{cur} of its current top-k intermediate results with results in N (see line 2, Algorithm 3). If its current result set coverage cov is greater than the defined threshold result set coverage cov', then the peer computes the improvement threshold $delta$ using the dynamic function H and subsequently the improvement impact imp (see lines 3-5, Algorithm 3). If imp is greater than or equal to $delta$ or if there are no more children' results to wait for, then the peer sends the set $T_{tosend} = T_{cur} \setminus T_{old}$ to its parent and subsequently sets T_{curr} to T_{old} (see lines 6-9, Algorithm 3). Recall that T_{cur} is the set of the current top-k intermediate results and T_{old} is the top-k intermediate results sent so far to its parent.

Algorithm 3: $Dtreat(k, T_{cur}, T_{old}, N, Func, cov, c\bar{o}v, H)$

```
input : k; T_cur; T_old; N; Func; cov, current local result set coverage; cōv,
        result set coverage threshold; H, a dynamic threshold function.
1  begin
2      T_cur = mergingSort_Topk(k, T_cur, N);
3      if (cov > cōv) then
4          delta = H(cov);
5          imp = Func(T_cur, T_old);
6          if ((imp ≥ delta) or all_Results()) then
7              T_tosend = T_cur \ T_old;
8              send_Parent(T_tosend, all_Results());
9              T_old = T_cur;
10         end
11     end
12 end
```

6 Dealing with peers failures in ASAP

One main characteristics of P2P systems is the dynamic behaviour of peers. It may happen that some peers leave the system (or fail) during the query processing. As a result peers may become inaccessible in the result bubbling up phase. In this section, we deal with this problem.

6.1 Absence of parent

In the query results bubbling up phase, each peer p bubbles up the results of peers in its subtree to its parent. It may happen that p's parent is inaccessible because it has left the system or failed. The question is which path to choose to bubble up p's intermediate results to the query originator. To deal with this problem a naive solution is that p sends its intermediate results directly to the query originator when p's parent is inaccessible. Recall that at the query forwarding phase the IP address and port of the query originator is communicated to all peers which have received the query. However the naive approach has some drawbacks. First, it may incur expensive merge of intermediates results at query originator which may be resource consuming. Second, by returning intermediate results of the peer whose parent is failed directly to the query originator, we reduce the capacity of peers to prune uninteresting intermediate results and this may increase significantly the volume of transferred data over the network.

Our solution to deal with the above mentioned problem is as following. Each peer p maintains locally for each active query q a list $QPath$ involving the addresses of peers (IP addresses and ports) in the path from the query originator to p in the q's forwarding tree. $QPath$ list is sorted by increasing positions of peers from the peer p in query q forwarding tree (the first item in this list is the parent of p, the second item is the grand parent of p, etc.). For constructing this list, each peer, including the query originator, adds its address to the query message before forwarding it to the neighbours. Thus, when a query message reaches a peer p, it contains the address of all parents of p.

In the phase of results bubble up when a peer detects that his parent (i.e first item in the list $QPath$) is inaccessible, the peer sends its new results to

the next peer in the list $QPath$ which is reachable. Another problem which may happen is that a peer may leave the system without being able to send to its parent the results received so far from its children, and this may have serious impact in the accuracy of final top-k results. To overcome this problem we adopt the following approach. During the results bubbling up phase, when a peer finds that its parent is unreachable, it sends its current top-k results to the next available peer in the list $QPath$. Although, this technique can increase the volume of data transferred in highly dynamic environment it may improve significantly the accuracy of top-k results.

6.2 Adjustment of Peer' local result set coverage

When computing the local result set coverage, we must take into account the fact that a peer may change parent when its direct parent becomes inaccessible. Indeed, not taking this into account will result in overestimation of peers' result sets coverage which may affect the value of the impact of intermediate results and thereby reducing the ability of peers to bubble up good quality results as soon as possible. In this section, we present our technique for adjusting the local result set coverage which is based on updating sets $\mathcal{C}_1$ and $\mathcal{C}_2$ which each p_i maintains for each top-k query q and for each child p_j. Recall that $\mathcal{C}_1$ is a set of pairs (p_j, a) where $a \in \mathbb{N}$ is the number of peers in the sub-tree of peer p_j and $\mathcal{C}_2$ a set of pairs (p_j, e) where $e \in \mathbb{N}$ is the total number of peers in the sub-tree of peer p_j which have already processed locally the query.

To help peers to have a good estimation of their result set coverage when some peers become inaccessible, we modify as follows our approach for peers failures management presented previously. Each peer p_i maintains for its parent p_j and for each active query the latest values of the estimation of number of peers which have already processed the query in the sub-tree and the number of peers in its sub-tree it has sent to p_j. In the results bubbling up phase, when p_j is inaccessible, p_i inserts into its answer message to its new parent p_k (the first accessible peer in $QPath$ list) the following information: 1) the new and the latest (sent to p_j) values of the number of peers and the number of peers which have already processed locally in the sub-tree; 2) the address of the peer which is before p_k in the list $QPath$ (this peer is one of child of p_k in the query forwarding tree) .

When a peer p_k receives an answer message of a query q from a peer p_j whose parent is inaccessible, it updates its estimation about the number of peers and the number of peers which have already processed locally q in the sub-tree of its direct child p_r which is declared as "inaccessible" by p_j (see lines 2-17, Algorithm 4). Then p_k activates a trigger to inform p_r (when it becomes accessible) that p_j is no longer in its sub-tree (see line 18, Algorithm 4).

7 Feedback Measures For Intermediate Results

Although it is important to provide good quality results as soon as possible to users, it is also interesting to associate "probabilistic guarantee" to the interme-

Algorithm 4: *result_Coverage_Adjustment(msg)*

```
   input : msg, an answer message of a query; C1; C2.
 1 begin
 2   if (change_Parent(msg) then
 3     qid = msg.getQueryID();
 4     sender = msg.getAnswerSender();
 5     a_l = getNbPeersSent(msg);
 6     e_l = getNbAnsPeersSent(msg);
 7     p = getLastPeerInacessible(msg);
 8     if (e_l > C2.get(p)) then
 9       x1 = C2.get(p);
10       y1 = C1.get(p);
11     else
12       x2 = C2.get(p) − e_l;
13       y2 = C1.get(p) − a_l;
14     end
15     C2.update(p, (x1+x2)/2);
16     y2 = C1.update(p, (y1+y2)/2);
17   end
18   propagateUpdate(queryId, Sender, p, a_l, e_l);
19 end
```

diate results allowing the user to know how far these results are from the final results. For example, we may wish to be able to give probabilistic guarantees, such as: "with probability γ, the current top-k results are likely to be the final top-k results". Our goal is to provide a mechanism to continuously compute these guarantees as results are bubbled up to the query originator. To do so, we compute two feedback measures which are returned to the user continuously: 1) the proportion of peers whose local results are already considered in the computation of the current top-k (we call this the proportion of contributor peers); 2) the probability of having the best k results in the current top-k results (we call this the stabilization probability). In this section, we present how these feedback measures can be computed.

Note that our goal is not to provide approximative top-k result with probabilistic guarantees where at the end of the query execution approximative top-k result set is returned to user with a probability showing how this result is far from the exact top-k result. In our work, probabilistic guarantees are computed continuously during the execution of the query on intermediate results as in [5]. Notice that the work presented in [5] considered centralized databases where it is easier to get some information on data stored (e.g data distribution or scores distribution, number of data stored, etc.) and to collect statistics during query processing, in order to provide probabilistic guarantees for top-k intermediate results. Therefore, the approach proposed in [5] cannot easily be applied for unstructured P2P system where data are completely distributed (i.e. there is no centralized catalog).

7.1 Stabilization probability

To be able to calculate the stabilization probability (i.e the probability that the current top-k is the exact top-k for a query q), we use the following information:

– the total number L of queried peers for the query q
– the total number M of data items shared by the L peers
– the number l of peers whose data items are taken into account in calculating the current top-k result of q
– the total number m of data items shared by the l peers

In Section 4.1, we presented how to estimate the parameter L and how to calculate l. Note that the calculation of m can be done by including in each answer message which a peer sends to its parent the number of data items which have already taken into account in the calculation of this response. However to be able to estimate M it is necessary to know the number l' of peers that are already known by the query initiator and the number m' of data items of those peers. The mechanism to calculate l' and m' works as follows. Each peer p_i maintains for each child p_j a set $\mathcal{C}_3$ of pairs (p_j, c, d) where c is the number of peers which p_i knows in the sub-tree whose root is p_j and d the number of data items shared by the c peers. At the beginning of query processing, each peer sets $c = 0$ and $d = 0$. In the phase of result bubbled up, when a child p_j in $\psi(p_i, q)$ wants to send results to p_i, it inserts in the answer message the couple of values $(\sum_{(p_j,c,d)\in\mathcal{C}_3} c, \sum_{(p_j,c,d)\in\mathcal{C}_3} d)$. Once p_i receives this message, it unpacks the message, gets these values and updates the set $\mathcal{C}_3$.

The total number M of data items of all queried peers is then estimated using Formula 6.

$$M = \|D(p_0)\| + m' + \frac{\|D(p_0)\| + m'}{l' + 1} * (L - l' - 1) \tag{6}$$

where $D(p_0)$ is the number of data items shared by the query originator.

By assuming that the data distribution over peers is uniform, the probability P_k^m of finding the k best data items in the current top-k result is:

$$P_k^m = \frac{C_{M-k}^{m-k}}{C_M^m} \tag{7}$$

If l peers over L have already bubbled up their local results to query originator, the probability of having m data items on these l peers is:

$$P_l^m = \begin{cases} 1 & if\ l = L \\ C_M^m \times (\frac{l}{L})^m \times (\frac{L-l}{L})^{M-m} & otherwise \end{cases} \tag{8}$$

Knowing that l peers have already bubbled up their local results to query originator, the probability of having at least k data items is given by:

$$P_l^{\geq k} = \sum_{m=k}^{M} P_l^m \tag{9}$$

To find all the k best results in those l peers there must be at least k data items on these l peers and all the best results must be owned by these l peers.

Thus the probability of having all the the top-k results in the current top-k result set is equal to:

$$P_l^{ktop} = \sum_{m=k}^{M} P_l^m \times P_k^m \tag{10}$$

To ensure a better estimation of the probability that the current top-k is the exact top-k in the case of peers failures, we have adopted the technique presented in Section 6 to readjust the estimation of all parameters used for calculating that probability.

7.2 Proportion of contributor peers

The proportion of contributor peers of a given current top-k results is the number of queried peers whose local results are already considered in the computation of that current top-k over the total number of queried peers. This proportion is equal to estimation of the query originator local result set coverage presented in Section 5. Thus, continuously we return the latter coverage to the user as the proportion of contributor peers.

7.3 Discussion

In some cases, it may happen that an unstructured P2P system is configured so that an issued query reaches all peers in the system (e.g by using very high ttl). In this case an efficient way to estimate the number of queried peers (i.e the network size) is to use gossip-based aggregation approach [19]. This approach relies on the following statement: if exactly one node of the system holds a value equal to 1, and all the other values are equal to 0, the average is $1/N$. The system size could thus be directly computed. To run this algorithm, an initiator should take the value equal to 1, and start gossiping; the reached nodes participate to the process by setting their value to 1. At each predefined cycle, each node in the network chooses one of its neighbors at random and swaps its estimation parameter (the network size and the number of shared data items). The contacted node does the same (push/pull heuristic of [19]). Both nodes then recompute their estimation as follows:

$$Estimation = \frac{Estimation + Neighbor's\_Estimation}{2}$$

By relying on gossip-based aggregation approach, we can also estimate the total number of data items shared by all peers in the system.

Notice that to provide correct estimations, this algorithm needs to wait a certain number of rounds to elapse before computing the size estimation; this period is the required time for the gossip to propagates in the whole overlay and for the values to converge. Notice that this method converge to the exact value in the stable system as demonstrated in [19]. Gossip protocols have been shown to provide exponentially fast convergence with low message transmission overhead as presented in [21].

8 Performance Evaluation

In this section, we evaluate the performance of ASAP through simulation using the PeerSim simulator [20]. This section is organized as follows. First, we describe our simulation setup, the metrics used for performance evaluation. Then, we study the effect of the number of peers and the number of results on the performance of ASAP, and show how it scales up. Next, we study the effect of the number of replicas on the performance of ASAP. We also study the effectiveness of our solution for providing probabilistic guarantees on the top-k results. After that, we study the effect of data distribution on the performance of ASAP. Finally, we investigate the effect of peers failures on the correctness of ASAP.

8.1 Simulation Setup

We implemented our simulation using the PeerSim simulator. PeerSim is an open source, Java based, P2P simulation framework aimed to develop and test any kind of P2P algorithm in a dynamic environment. It consists of configurable components and it has two types of engines: cycle-based and event-driven engine. PeerSim provides different modules that manage the overlay building process and the transport characteristics.

We conducted our experiments on a machine with a 2.4 GHz Intel Pentium 4 processor and 2GB memory. The simulation parameters are shown in Table 1. We use parameter values which are typical of P2P systems [15]. The latency between any two peers is a normally distributed random number with mean of 200 ms. Since users are usually interested in a small number of top results, we set $k = 20$ as default value. In our experiments we vary the network size from 1000 to 10000 peers. In order to simulate high heterogeneity, we set peers' capacities in our experiments, in accordance to the results in [15]. This work measures the peers capacities in the Gnutella system. Based on these results, we generate around 10% of low-capable, 60% of medium-capable, and 30% of high-capable peers. The high-capable peers are 3 times more capable than medium-capable peers and still 7 times more capable than low-capable ones.

In the context of our simulations each peer in the P2P system has a table $\mathcal{R}(data)$ in which attribute $data$ is a real value. The number of rows of $\mathcal{R}$ at each peer is a random number uniformly distributed over all peers greater than 1000 and less than 20000. Unless otherwise specified, we assume only one copy of each data item in our system (i.e. no data replication). We also ensure that there are not two different data items with the same score. In all our tests, we use the following simple query, denoted by q_{load} as workload:
`SELECT` val `FROM` $\mathcal{R}$ `ORDER BY` $F(\mathcal{R}.data, val)$ `STOP AFTER` k
The score $F(\mathcal{R}.data, val)$ is computed as:

$$\frac{1}{1 + |\mathcal{R}.data - val|}$$

In our simulation, we compare ASAP with Fully Distributed (FD) [2], a baseline

Parameters	Values
Latency	Normally distributed random number, $Mean = 200$ ms, $Variance = 100$
Number of peers	10,000 peers
Average degree of peers	4
ttl	9
k	20
Number of replicas	1

Table 1. Simulation parameters.

approach for top-k query processing in unstructured P2P systems which works as follows. Each peer that receives the query, executes it locally (i.e. selects the k top scores), and waits for its children's results. After receiving all its children score-lists, the peer merges its k local top data items with those received from its children and selects the k top scores and sends the result to its parent.

In our experiments, to evaluate the performance of ASAP comparing to FD, we use the following metrics:

(i) **Cumulative quality gap:** As defined in Section 3, is the sum of the quality difference between intermediate top-k result sets received until the stabilization time and the final top-k result set.
(ii) **Stabilization time:** We report on the stabilization time, the time of receiving all the final top-k results.
(iii) **Response time:** We report on the response time, the time the query initiator has to wait until the top-k query execution is finished.
(iv) **Communication cost:** We measure the communication cost in terms of number of answer messages and volume of data which must be transferred over the network in order to execute a top-k query.
(v) **Accuracy of results:** We define the accuracy of results as follows. Given a top-k query q, let V be the set of the k top results owned by the peers that received q, let V' be the set of top-k results which are returned to the user as the response of the query q. We denote the accuracy of results by ac_q and we define it as

$$ac_q = \frac{\|V \cap V'\|}{\|V\|}$$

(iv) **Total number of results:** We measure the total number of results as the number of results received by the query originator during query execution.

In our experimentation, we perform 30 tests for each experiment by issuing q_{load} 20 different times and we report the average of their results. Due to space limitations, we only present the main results of ASAP's dynamic threshold-based approaches denoted by ASAP-Dscore and ASAP-Drank. ASAP-Dscore uses a score-based improvement impact and ASAP-Drank a rank-based improvement

impact. ASAP's dynamic threshold-based approaches have proved to be better than ASAP's static threshold-based approaches without being expensive in communication cost. In our all experiments, for ASAP-Dscore approach we use $H(x) = -0.2x + 0.2$ as dynamic threshold function and 0 as peer's local result set coverage threshold. In the case of Asap-Drank, we use $H(x) = -0.5x + 0.5$ as dynamic threshold function and 0.05 as peer's local result set coverage threshold.

8.2 Performance Results

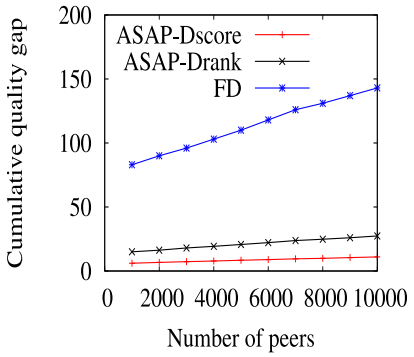
Effect of number of peers We study the effect of the number of peers on the performance of ASAP. For this, we ran experiments to study how cumulative quality gap, stabilization time, number of answer messages, volume of transferred data, number of intermediate results and response time increase with the addition of peers. Note that the other simulation parameters are set as in Table 1.

Figure 2(a) and 2(b) show respectively how cumulative quality gap and stabilization time increase with the number of peers. The results show that the cumulative quality gap of ASAP-Dscore and ASAP-Drank is always much smaller than that of FD, which means that ASAP returns quickly high quality results. The results also show that the stabilization time of ASAP-Dscore is always much smaller that of ASAP-Drank and that of FD. The reason is that ASAP-Dscore is score sensitive, so the final top-k results are obtained quickly.

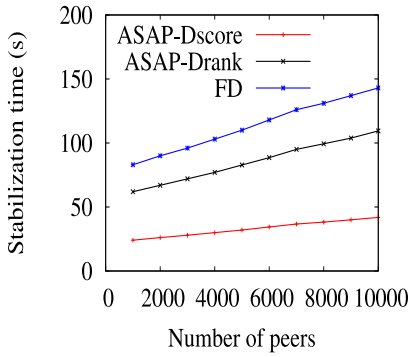
Figure 2(c) shows that the total number of results received by the user increases with the number of peers in the case of ASAP-Dscore and ASAP-Drank while it is still constant in the case of FD. This is due to the fact that FD does not provide intermediate results to users. The results also show that the number of results received by the user in case of ASAP-Dscore is smaller than that of ASAP-Drank. The main reason is that ASAP-Dscore is score sensitive in contrast to ASAP-Drank.

Figure 2(d) and Figure 2(e) show that the number of answer messages and volume of transferred data increase with the number of peers. The results show that the number of answer messages and volume of transferred data of ASAP-Drank are always higher than those of ASAP-Dscore and FD. The results also show that the differences between ASAP-Dscore and FD's number of answer messages and volume of transferred data are not significant. The main reason is that ASAP-Dscore is score sensitive in contrast to ASAP-Drank. Thus, only high quality results are bubbled up quickly.

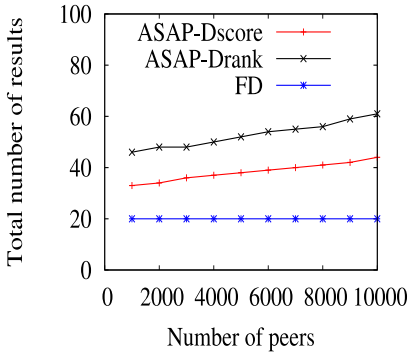
Figure 2(f) shows how response time increases with increasing the numbers of peers. The results show that the difference between ASAP-Dscore and FD response time is not significant. The results also show that the difference between ASAP-Drank and FD's response time increases slightly in favour of ASAP-Drank as the number of peers increases. The reason is that ASAP-Drank induces more network traffic than ASAP-Dscore and FD.



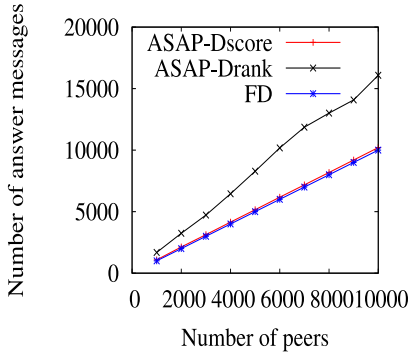
(a) Cumulative quality gap vs. Number of peers



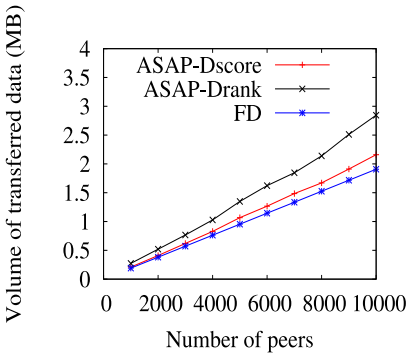
(b) Stabilization time vs. Number of peers



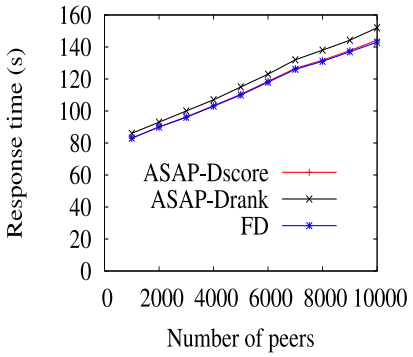
(c) Total number of results vs. Number of peers.



(d) Number of answer messages vs. Number of peers

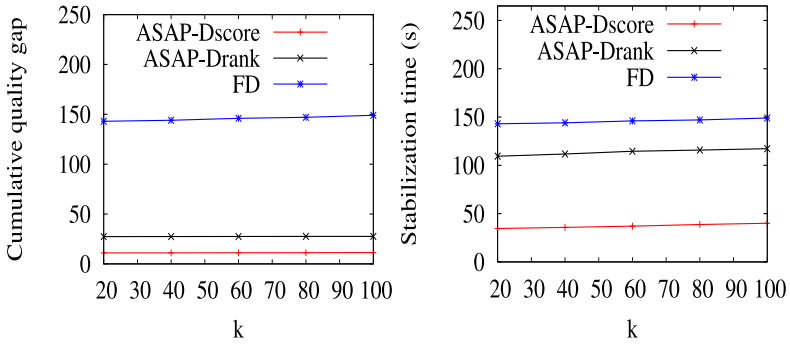


(e) Volume of transferred data vs. Number of peers

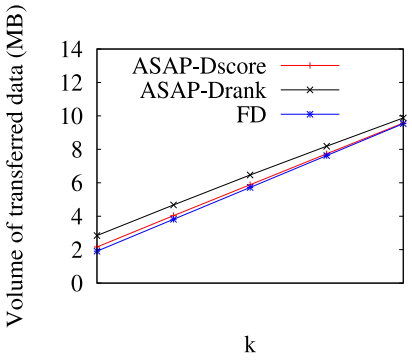


(f) Response time vs. Number of peers

Fig. 2. Impact of number of peers on ASAP performance

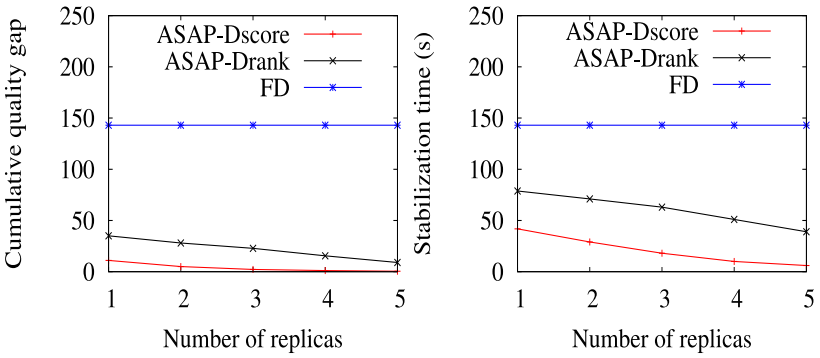


(a) Cumulative quality gap vs. k (b) Stabilization time vs. k



(c) Volume of transferred data vs. k

Fig. 3. Impact of k on ASAP performance



(a) Cumulative quality gap vs. Number of replicas (b) Stabilization time vs. Number of replicas

Fig. 4. Impact of data replication on ASAP performance

Effect of k We study the effect of k, i.e. the number of results requested by the user, on the performance of ASAP. Using our simulator, we studied how

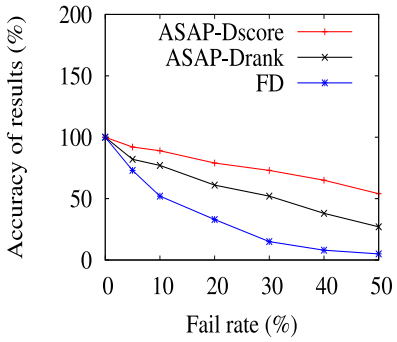


Fig. 5. Accuracy of results vs. fail rate

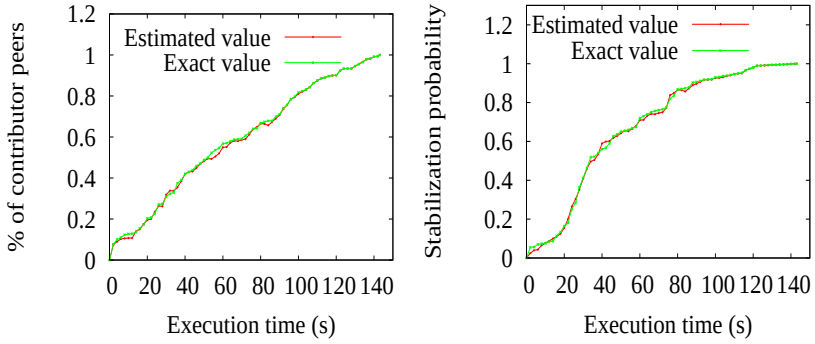
cumulative quality gap, stabilization time and volume of transferred data evolve while increasing k from 20 to 100, with the other simulation parameters set as in Table 1. The results (see Figure 3(a), Figure 3(b)) show that k has very slight impact on cumulative quality gap and stabilization time of ASAP-Dscore and ASAP-Drank. The results (see Figure 3(c)) also show that by increasing k, the volume of transferred data of ASAP-Dscore and ASAP-Drank increase less than that of FD. This is due to the fact that ASAP-Dscore and ASAP-Drank prune more intermediate results when k increases.

Data replication Replication is widely used in unstructured P2P systems to improve search or achieve availability. For example, modern unstructured overlays like BubbleStorm [29] use large number of replicas for each object placed in the overlay to improve their search algorithms.

We study the effect of the number of replicas, which we replicate for each data (uniform replication strategy [14]), on the performance of ASAP. Using our simulator, we studied how cumulative quality gap and stabilization time evolve while increasing the number of replicas, with the other simulation parameters set as in Table 1. The results (see Figure 4(a) and Figure 4(b)) show that increasing the number of replicas for ASAP and FD decrease ASAP-Dscore and ASAP-Drank's cumulative quality gap and stabilization time. However, FD's cumulative quality gap and stabilization time are still constant. The reason is that ASAP returns quickly the results having high quality in contrast to FD which returns results only at the end of query execution. Thus, if we increase the number of replicas, ASAP finds quickly the results having high scores.

Effectiveness of our solution for computing "probabilistic guarantees" In this section, we study the effectiveness of the proposed solution in Section 7 for computing the probabilistic guarantees, by comparing them with optimal values. For this, we ran experiments to study how our probabilistic guarantees values evolve comparing to the optimal (i.e real) values during the query execution in

the case of ASAP-Dscore. Figure 6(a) and Figure 6(b) show that the difference between our probabilistic guarantees values and the exact values is very slight. The results also show that our probabilistic guarantees values converge to exact values and this before the end of the execution of the query execution. This means that our solution provides reliable guarantees for the user on the intermediate results.



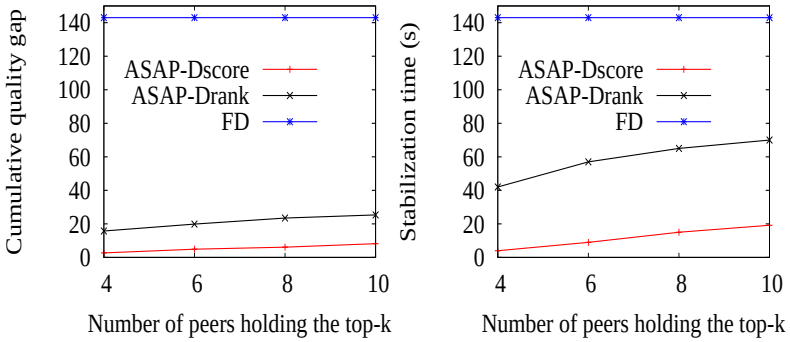
(a) Proportion of contributor peers vs. Execution time (s)

(b) Stabilization probability vs. Execution time (s)

Fig. 6. Effectiveness of our solution for computing probabilistic guarantees

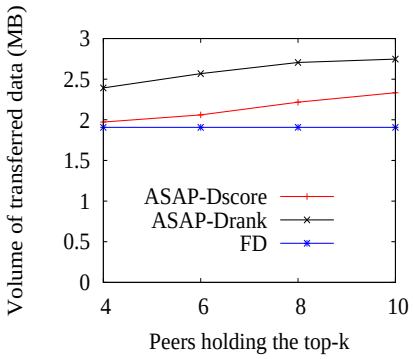
Data distribution In this section, we study the effect of data distribution on the performance of our top-k query processing solution. Often relevant data items are grouped together, stored on a group of neighbouring peers. If these groups of peers have some good data objects for top-k, they become the sweet region in the network that can contribute a lot to a final top-k. To study the effect of data distribution on the performance of ASAP, we randomly distribute the top-k data items of our test bed queries respectively on 4, 6, 8 and 10 peers of P2P system and the other data (i.e which are not in the top-k results) uniformly over all the peers of the system. Using our simulator, we studied how cumulative quality gap, stabilization time and volume of transferred data evolve while only 4, 6, 8 and 10 peers of the P2P system store the top-k results with the other simulation parameters set as in Table 1. The results (see Figure 7(a) and Figure 7(b)) show that ASAP can take advantage of grouped data distribution to provide quickly high quality results to users in contrast to FD. The results (see Figure 7(c)) also show that in the case of ASAP, the higher the top-k data items are grouped together, the smaller is the volume of transferred data over the network, while this volume is constant in the case of FD.

Effect of peers failures In this section, we investigate the effect of peers' failures on the accuracy of top-k results of ASAP. In our tests, we vary the value



(a) Cumulative quality gap vs. Number of peers holding the top-k results

(b) Stabilization vs. Number of peers holding the top-k results



(c) Volume of transferred data vs. Number of peers holding the top-k results

Fig. 7. Impact of data distribution on ASAP performance

of fail rate and investigate its effect on the accuracy of top-k results. Figure 5 shows accuracy of top-k results for ASAP-Dscore, ASAP-Drank and FD while increasing the fail rate, with the other parameters set as in Table 1. Peers' failures have less impact on ASAP-Dscore and ASAP-Drank than FD. The reason is that ASAP-Dscore and ASAP-Drank return the high-score results to the user as soon as possible. However, when increasing the fail rate in FD, the accuracy of top-k results decreases significantly because some score-lists are lost. Indeed, in FD, each peer waits for results of its children so in the case of a peer failure, all the score-lists received so far by that peer are lost.

9 Related Work

Efficient processing of top-k queries is both an important and hard problem that is still receiving much attention. Several papers have dealt with top-k query pro-

cessing in centralized database management systems [9, 18, 27, 24]. In distributed systems [10, 16, 7, 31, 33], previous work on top-k processing has focused on vertically distributed data over multiple sources, where each source provides a ranking over some attributes. Some of the proposed approaches, such as recently [3], try to improve some limitations of the Threshold Algorithm (TA) [13]. Following the same concept, there exist some previous work for top-k queries in P2P over vertically distributed data. In [8], the authors propose an algorithm called "Three-Phase Uniform Threshold" (TPUT) which aims at reducing communication cost by pruning away intelligible data items and restricting the number of round-trip messages between the query originator and other nodes. Later, TPUT was improved by KLEE [22] that uses the concept of bloom filters to reduce the data communicated over the network upon processing top-k queries. It brings significant performance benefits with small penalties in result precision. However, theses approaches assume that data is vertically distributed over the nodes whereas we deal with horizontal data distribution.

For horizontally distributed data, there has been little work on P2P top-k processing. In [2], the authors present FD, a fully distributed approach for top-k query processing in unstructured P2P systems. We have briefly introduced FD in section 8.1.

PlanetP [11] is the content addressable publish/subscribe service for unstructured P2P communities up to ten thousand peers. PlanetP uses a gossip protocol to replicate global compact summaries of content (term-to-peer mappings) which are shared by each peer. The top-k processing algorithm works as follows. Given a query q, the query originator computes a relevance ranking (using the global compact summary) 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. However, in a large P2P system, keeping up-to-date the replicated index is a major problem that hurts scalability.

In [6], the authors present an index routing based top-k processing technique for super-peer networks organized in an HyperCuP topology which tries to minimize the number of transfer data. The authors use statistics on queries to maintain the indexes built on super-peers. However, the performance of this technique depends on the query distribution.

In [32], the authors present SPEERTO, a framework that supports top-k query processing in super-peer networks by using a skyline operator. In SPEERTO, for a maximum of K, denoting an upper bound on the number of results requested by any top-k query ($k \leq K$), each peer computes its K-skyband as a pre-processing step. Each super peer maintains and aggregates the K-skyband sets of its peers to answer any incoming top-k query. The main drawback of this approach is that each join or leave of peer may induce the recomputing of all super-peers K-skyband. Although these techniques are very good for super-peers systems, they cannot apply efficiently for unstructured P2P systems, since there may be no peer with high reliability and computing power.

Zhao *et al.* [34] use a result caching technique to prune network paths and answer queries without contacting all peers. The performance of this technique

depends on the query distribution. They assume acyclic networks, which is restrictive for unstructured P2P systems.

10 Conclusion

In this paper we deal with as-soon-as-possible top-k query processing in P2P systems. We proposed a formal definition for as-soon-as-possible top-k query processing by introducing two novels notions: stabilization time and cumulative quality gap. We presented ASAP, a family of algorithms which uses a threshold-based scheme that considers the score and the rank of intermediate results to return quickly the high quality results to users. We validated ASAP through implementation and extensive experimentation. The results show that ASAP significantly outperforms baseline algorithms by returning final top-k result to users in much better times. Finally, the results demonstrate that in the presence of peers' failures, ASAP provides approximative top-k results with good accuracy, unlike baseline algorithms.

References

1. R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. *Global Data Management*, chapter Design and Implementation of Atlas P2P Architecture. IOS Press, first edition, 2006.
2. R. Akbarinia, E. Pacitti, and P. Valduriez. Reducing network traffic in unstructured p2p systems using top-k queries. *Distributed and Parallel Databases*, 19(2-3):67–86, 2006.
3. R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *Proceedings of Int. Conf. on Very Large Data Bases (VLDB)*, pages 495–506, 2007.
4. S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.
5. B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top-k algorithms. In *Proceedings of Int. Conf. on Very Large Data Bases (VLDB)*, pages 914–925, 2007.
6. W.-T. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive distributed top k retrieval in peer-to-peer networks. In *Proceedings of Int. Conf. on Data Engineering (ICDE)*, pages 174–185, 2005.
7. N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *Proceedings of Int. Conf on Data Engineering (ICDE)*, pages 369–380, 2002.
8. P. Cao and Z. Wan. Efficient top-k query calculation in distributed networks. In *Proceedings of Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 206–215, 2004.
9. S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proceedings of Int. Conf. on Very Large Databases (VLDB)*, pages 397–410, 1999.
10. S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Transactions on Knowledge Data Engineering*, 16(8):992–1009, 2004.

11. 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. In *Proceedings of IEEE Int. Symp. on High-Performance Distributed Computing (HPDC)*, pages 236–249, 2003.
12. W. K. Dedzoe, P. Lamarre, R. Akbarinia, and P. Valduriez. Asap top-k query processing in unstructured p2p systems. In *Proceedings of IEEE Int. Conf on Peer-to-Peer Computing (P2P)*, pages 187–196, 2010.
13. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 102–113, 2001.
14. G. Feng, Y. Jiang, G. Chen, Q. Gu, S. Lu, and D. Chen. Replication strategy in unstructured peer-to-peer systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.
15. P. K. Gummadi, S. Saroiu, and S. D. Gribble. A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems. *Computer Communication Review*, 32(1):82, 2002.
16. U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *Proceedings of Int. Conf. on Very Large DataBases (VLDB)*, pages 419–428, 2000.
17. K. Hose, M. Karnstedt, K.-U. Sattler, and D. Zinn. Processing top-n queries in p2p-based web integration systems with probabilistic guarantees. In *Proceedings of International Workshop on web and databases (WebDB)*, pages 109–114, 2005.
18. V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: a system for the efficient execution of multi-parametric ranked queries. In *Proceedings of ACM. Int Conf. on Management of Data (SIGMOD)*, pages 259–270, 2001.
19. M. Jelasity and A. Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Int. Conference on Distributed Computing Systems (ICDCS)*, pages 102–109, 2004.
20. M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. The Peersim simulator. `http://peersim.sf.net`.
21. D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Symposium on Foundations of Computer Science (FOCS)*, pages 482–491, 2003.
22. S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *Proceedings of Int. Conf. on Very Large Data Bases (VLDB)*, pages 637–648, 2005.
23. B. C. Ooi, Y. Shu, and K.-L. Tan. Relational data sharing in peer-based data management systems. *SIGMOD Record*, 32(3):59–64, 2003.
24. L. Qin, J. X. Yu, and L. Chang. Diversifying top-k results. *PVLDB*, 5(11):1124–1135, 2012.
25. L. Ramaswamy, J. Chen, and P. Parate. Coquos: Lightweight support for continuous queries in unstructured overlays. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2007.
26. S. Schmid and R. Wattenhofer. Structuring unstructured peer-to-peer networks. In *Proceedings of IEEE Int. Conf. on High Performance Computing (HiPC)*, pages 432–442, 2007.
27. M. Shmueli-Scheuer, C. Li, Y. Mass, H. Roitman, R. Schenkel, and G. Weikum. Best-effort top-k query processing under budgetary constraints. In *Proceedings of Int. Conf. on Data Engineering (ICDE)*, pages 928–939, 2009.

28. I. Tatarinov, Z. G. Ives, J. Madhavan, A. Y. Halevy, D. Suciu, N. N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The piazza peer data management project. *SIGMOD Record*, 32(3):47–52, 2003.
29. W. W. Terpstra, J. Kangasharju, C. Leng, and A. P. Buchmann. Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In *SIGCOMM*, pages 49–60, 2007.
30. D. Tsoumakos and N. Roussopoulos. Analysis and comparison of p2p search methods. In *Proceedings of Int. Conf. on Scalable Information Systems (Infoscale)*, page 25, 2006.
31. A. Vlachou, C. Doulkeridis, and K. Nørvåg. Distributed top-k query processing by exploiting skyline summaries. *Distributed and Parallel Databases*, 30(3-4):239–271, 2012.
32. A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis. On efficient top-k query processing in highly distributed environments. In *Proceedings of ACM. Int Conf. on Management of Data (SIGMOD)*, pages 753–764, 2008.
33. M. Ye, W.-C. Lee, D. L. Lee, and X. Liu. Distributed processing of probabilistic top-k queries in wireless sensor networks. *IEEE Trans. Knowl. Data Eng.*, 25(1):76–91, 2013.
34. K. Zhao, Y. Tao, and S. Zhou. Efficient top-k processing in large-scaled distributed environments. *Data and Knowledge Engineering*, 63(2):315–335, 2007.