Collaborative Personalized Top-k Processing

XIAO BAI, Yahoo! Research Barcelona
RACHID GUERRAOUI, EPFL
ANNE-MARIE KERMARREC, INRIA Rennes Bretagne-Atlantique
VINCENT LEROY, Yahoo! Research Barcelona

This article presents P4Q, a fully decentralized gossip-based protocol to personalize query processing in social tagging systems. P4Q dynamically associates each user with social acquaintances sharing similar tagging behaviors. Queries are gossiped among such acquaintances, computed on-the-fly in a collaborative, yet partitioned manner, and results are iteratively refined and returned to the querier. Analytical and experimental evaluations convey the scalability of P4Q for top-*k* query processing, as well its inherent ability to cope with users updating profiles and departing.

Categories and Subject Descriptors: H.2.4 [**Database Systems**]: Systems—*Distributed databases; query processing*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Peer-to-peer networks, top-k processing, personalization, gossip

ACM Reference Format:
Bai, X., Guerraoui, R., Kermarrec, A.-M., and Leroy, V. 2011. Collaborative personalized top-k processing. ACM Trans. Datab. Syst. 36, 4, Article 26 (December 2011), 38 pages.
DOI = 10.1145/2043652.2043659 http://doi.acm.org/10.1145/2043652.2043659

1. INTRODUCTION

The Web 2.0 revolution has transformed the Internet from a read-only infrastructure to an active read-write platform. The content of collaborative tagging systems, such as delicious, Flickr and YouTube, is generated by the users themselves, who annotate items with freely chosen keywords. Such collaborative tagging systems represent huge mines of information. Yet, exploring these mines is challenging because of the unstructured nature of tagging and the lack of any fixed ontology. Clearly, the user freedom to choose tags is key to the popularity of these systems but this freedom is also a significant source of ambiguities in the search process.

An appealing way to reduce the exploration space in collaborative tagging systems is to personalize the search by exploiting information from the social acquaintances of the user, typically users that exhibit similar tagging behaviors. If a computer scientist searches "matrix" in Google, for example, she is probably seeking some mathematical notions, but the first several pages returned from Google are all about the movie *Matrix*.

This work is supported by the ERC Starting Grant, under the project GOSSPLE, number 204742.
This work was carried out during X. Bai and V. Leroy's stay in INRIA Rennes, France.
Authors' addresses: X. Bai (corresponding author), Yahoo! Research Barcelona, Spain; email: baixiao@msn.com; R. Guerraoui, School of Computer and Communication Sciences, EPFL, Switzerland; A.-M. Kermarrec, INRIA Rennes Bretagne-Atlantique, France; V. Leroy, Yahoo! Research Barcelona, Spain.

DOI 10.1145/2043652.2043659 http://doi.acm.org/10.1145/2043652.2043659

In contrast, a Keanu Reeves fan may just look for this movie. Personalization based on user affinities has the potential to disambiguate these situations.[1]

Several personalized approaches have been proposed to leverage social networks in search procedures [Mislove et al. 2006]. So far, however, these approaches have focused mainly on explicit social networks, that is, social networks established a priori, independently of the tagging profiles (e.g., Facebook). We argue for improving the information retrieval quality by exploiting the implicit user-centric correlation in shared interests. The motivation stems from the observation that people you might not know, but with whom you share many interests, can be very helpful when searching the Web. The effectiveness of implicit social networks over explicit social networks for improving the search result quality is confirmed in Bender et al. [2007]. However, the necessary mechanisms to discover and maintain the implicit social networks are not investigated in that work.

In this article, we address the problem of leveraging implicit social acquaintances in ranking the results of top-*k* queries in a large-scale collaborative tagging system. We focus on the design of an efficient protocol to support personalized query processing in peer-to-peer systems. This went through dealing with the large amount of information that needs to be maintained per user, as well as the dynamic nature of the system with users who continuously join or leave and periodically change their profiles by tagging new items.

The motivation stems from the fact that peer-to-peer solutions inherently circumvent the danger of central authorities abusing the information at their disposal, for example, exploiting the user profiles for commercial purposes. In a peer-to-peer system, information is distributed across users, avoiding the Big Brother syndrome. Another advantage of peer-to-peer solutions is that they scale naturally. The increase of users automatically brings the system more capability to store the information and process the queries.[2]

A natural, fully decentralized solution would consist for each user to locally store and maintain her (implicit) social network, enabling thereby efficient top-*k* query computation. Yet, this would require every user to store all profiles of her acquaintances: these would then be massively replicated and hence hard to maintain. Not surprisingly, and as shown in Bai et al. [2009], several hundreds of profiles are needed to return accurate results in a system of only 10,000 users. Maintaining all necessary profiles in a real system of several millions of users seems simply inadequate.

At the other extreme, a storage-effective strategy would consist for each user to store and maintain only her own profile and seek other profiles on-the-fly, that is, whenever a query is to be processed. Clearly, this optimizes the storage and maintenance issues but might induce a large number of messages and a large latency if profiles of acquaintances are to be consulted at query time. In addition, the profiles of temporarily disconnected users would be unavailable which, in turn, might significantly hamper the accuracy of the query processing.

In this article, we propose P4Q (*Performance-aware Personalized Peer-to-Peer Query processing*), a *bimodal, gossip-based* solution to personalize the query processing. P4Q does not rely on any central server: users periodically maintain their networks of social acquaintances by gossiping among each other and computing the proximity between

[1]Items concerning both mathematics and movies could be proposed if a computer scientist also turns out to be a Keanu Reeves fan. Yet, if a mathematician suddenly begins to issue queries for the movie *Matrix*, a default search mechanism would be activated to provide the mainstream results. For instance, such queries can be easily redirected to the state-of-the-art commercial search engines. Our work focuses on the personalization aspect.

[2]Cloud-based solutions also scale well for dealing with large amounts of data, but require heavy investments and render the entire system at the mercy of the cloud provider.

tagging profiles. Every user maintains her social network; namely a set of IDs called the *personal network*. A user, however, only locally stores a limited subset of profiles, typically those of very similar users. The number of profiles stored at each user is chosen according to the storage capability of that user.

The maintenance of the personal network is performed in a *lazy* gossip mode, at a fairly low frequency to avoid overloading the network. To limit the bandwidth consumption, users first exchange digests of their profiles, encoded in Bloom filters, to estimate the proximity between their profiles.[3] They only exchange the whole profile when the similarity appears significant.

The querying scheme itself is based on an *eager* mode of the gossip protocol, that is, with an increased frequency, and is biased towards social acquaintances. Every query is first computed locally, based on the set of stored profiles, providing an immediate partial result to the user. Then the query, together with the list of profiles needed to compute it, is gossiped and computed collaboratively. The query is gossiped first to the closest acquaintances and further away according to the social proximity between user profiles, iteratively refining the results. Each user reached by the query locally computes her share of the query based on the relevant profiles stored locally, and then gossips the query further. The results are thus iteratively refined in a number of gossip cycles, harvesting relevant information at each step, and displayed directly at the querier. As the number of partial results to merge varies with time, we use a variant of the NRA algorithm (No Random Access [Fagin 2002]) to retrieve the *k* most relevant items from the partial results. The consistency of the top-*k* items is ensured by a version field associated to each user's profile. A partitioning technique prevents the users from biasing the results by computing queries against redundant profiles.

Gossiping the query: (i) avoids saturating the network by contacting all the users in the personal network at the same time, and (ii) refreshes the part of the network originating from the querier, generating a specific wave of refreshments in the personalization process. The user can, at any time, consult the results of the queries and decide whether these are satisfactory enough. As we will show through our experiments, only a few gossip cycles are sufficient to compute very good results.

We evaluate P4Q both analytically and experimentally. The analysis shows that the query processing time in gossip cycles can be approximated with $O(\log_2 L)$, where L is the number of profiles in a user's personal network that contribute to the query processing but are not stored by her. The analysis also bounds the number of messages incurred by the query propagation and partial results transmission. Our experimental evaluations confirm the analytical results. We evaluated P4Q using the *PeerSim* [Jelasity et al. 2004; Montresor and Jelasity 2009] simulator with a real dataset crawled in January 2009 from delicious involving 10,000 users. We considered several storage scenarios. We show that even if each user stores only 10 profiles in her personal network, top-*k* queries can be accurately satisfied within 10 gossip cycles, corresponding to 50 seconds with an eager mode running every 5 seconds. We highlight the trade-off between the user's expectation on query results, the latency of the response, and the space availability. Running the lazy mode every minute, even if all users simultaneously change their profiles, in half an hour, 95% of the stored information is updated. Meanwhile, P4Q incurs acceptable overload in terms of bandwidth consumption: 7.6Kbps are sufficient to maintain the personal network and 91Kbps are sufficient to compute a query. P4Q is also robust against user departures: a massive leave of 50% users impacts the quality by only 10%.

[3]To ensure a good accuracy of this estimation, a Bloom filter is based on both the tags and the items contained in a user profile. The size of a Bloom filter is dynamically adapted to the size of the encoded profile.

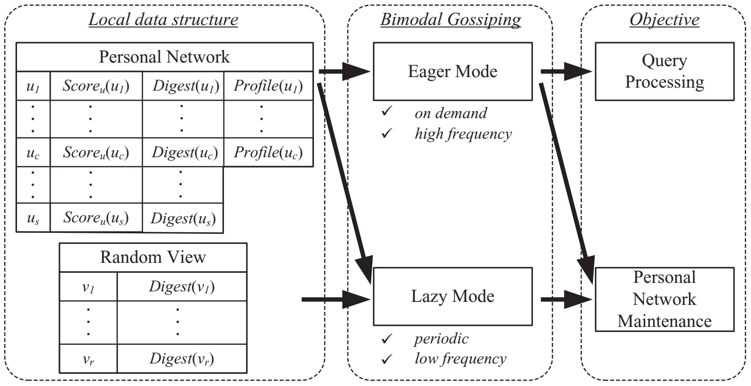


Fig. 1. System model.

To summarize, we present, for the first time, a decentralized technique to perform personalized search queries using implicit user affinities. Whereas we use standard techniques to compute profile proximity (number of common item-tag) and rank queries (NRA), P4Q is generic in the sense that alternatives (metrics and ranking algorithms) could be used. We discuss this flexibility in Section 3.5.1.

The rest of the article is organized as follows. Section 2 describes our P4Q protocol and analyses its behavior. Section 3 presents our experimental results. Section 4 concludes our work by discussing related work.

2. THE P4Q PROTOCOL

2.1. System Model and Data Structures

We consider a collaborative system as an information space, where U denotes the set of users, I contains the items in the system, and T is the set of all related tags. $Tagged_{u_x}(i, t, d)$ captures the fact that user u_x tagged item i with tag t at time d. The profile of user u_x is described as a set of her tagging actions, that is,

$$Profile(u_x) = \{\langle i, t, d\rangle | Tagged_{u_x}(i, t, d)\}.$$

Each profile $Profile(u_x)$ is associated to a version field, denoted as $Version(u_x)$, indicating its most recent modification time. In other words, once user u_x tags an item with a tag at time d, the version of her profile $Version(u_x)$ is updated to d. This version field is used to maintain the consistency of the query results during the processing. The network is modeled as a directed graph where each node corresponds to a user and an edge represents a link between two users. When there is a directed edge from user u_x to user u_y, u_y is considered as a neighbor of u_x. For the simplicity of presentation, we use the term user to mean its associated underlying machine and generally refer to the canonical user as "she."

In P4Q, except for her own profile, a user maintains two data structures: a personal network and a random (and dynamic) view of the network (Figure 1).

Personal Network. The personal network of user $u_x \in U$ is a set of s neighbors having the most similar interests with her, noted as $Network(u_x)$. This requires to compute a *distance* between users: u_x maintains a similarity score, denoted as $Score_{u_x}(u_y)$, for each neighbor u_y in $Network(u_x)$, which reflects such distance by quantifying the degree of similarity between u_x and u_y. In this article, we define the score as the number of common tagging actions in two users' profiles, that is,

$$Score_{u_x}(u_y) = |\{\langle i, t\rangle | Tagged_{u_x}(i, t, *) \wedge Tagged_{u_y}(i, t, *)\}|.$$

Here we use the term "*common tagging action*" to stand for the pair $\langle i, t\rangle$, where the same item i was tagged by both u_x and u_y with the same tag t, regardless of the time it has been tagged. We do not distinguish the term "tagging action" between the pair $\langle i, t\rangle$ and the triple $\langle i, t, d\rangle$ for the ease of presentation.

This similarity metric is inspired from those of Amer-Yahia et al. [2008b], where the tagging actions are used to model user interests in collaborative tagging systems and the number of shared tagging actions is shown to be positively correlated to the number of similar interests. As a tag can be used for different items and an item may receive different tags from different users, the metric we use accounts for the users' preferences on both topics (tags) and specific objects (items). The higher the score, the more interests are shared between u_x and u_y. In fact, the distance is application-specific and P4Q is *independent* of the way similarity is defined as we will see in Section 3.5.1.

The query results of user u_x depend only on the profiles of the neighbors in her personal network. This means that the relevance of an item for a query issued by a user is not computed based on the whole set of users but only the set of users restricted to her personal network. To guarantee the effectiveness of the query processing, the size of the personal network s should be relatively large. In order to maintain the local storage in reasonable bounds as well as keep the stored profiles up-to-date, only the profiles of the c neighbors u_y having the highest $Score_{u_x}(u_y)$ are stored. Note that users may adjust c depending on their expectation on the query results (with respect to latency and accuracy) and their storage availabilities. Typically, the larger c, the more accurate the results obtained by computing the query locally.

In order to limit the overhead of the protocol, a digest of profile ($Digest(u_y)$) is also stored along with each neighbor u_y in the personal network. A digest is a compact summary of a user's profile encoded using a Bloom filter [Bloom 1970]. The digests are used in P4Q to estimate the similarity between users.

Random View. Each user also maintains a set of r neighbors, called a random view, selected uniformly at random from the whole network and continuously renewed. These users are used to ensure that the network remains connected [Jelasity et al. 2007]. In addition, this enables the discovery of new similar neighbors. A digest of profiles is also stored for each neighbor so that they may be considered for the personal network.

Profile Digest. Since transferring the whole profile of a user may be bandwidth consuming, in P4Q an upper bound of the similarity between users is first computed based on the digests of profiles. This avoids transferring unnecessarily entire profiles. P4Q relies on Bloom filters to generate a compact representation of the profiles. Once a digest is generated according to a profile, the same version as the profile is assigned to it. A Bloom filter [Bloom 1970] is a space-efficient probabilistic data structure that is used to test the presence of an element in a set. An empty Bloom filter is a bit array of m bits, all set to 0. An element is added to a Bloom filter using a set of hash functions to determine the positions of this element in the Bloom filter. The bits at the corresponding positions are then set to 1. To query the presence of an element in a set, the same hash functions are used. The element is considered present if all the resulted bits have been set to 1. The main advantage of Bloom filters is that they do not generate any false negative, that is, if an element is in the set, its presence will be detected. Yet, an answer may be false positive as all the concerned bits can be set to 1 due to the insertion of other elements. P4Q relies on Bloom filters to assess the similarity score between users encountered during the personal network maintenance. As a consequence, using Bloom filters, P4Q may overestimate a score due to the false positive answers. Yet, should an estimation qualify a user to belong to the personal network, more information would be transmitted to obtain the exact similarity, as we will see in Section 2.2.1. In fact this can be controlled by adjusting the size of the Bloom filter according to the number of elements to insert.

A profile of a user consists of the tagging actions of this user, namely which item is annotated with which tag by this user. As a result, the information on tags, items, or both could be used to derive the profile digest from a profile.

The profile digest can be a Bloom filter of items (*ItemBloom*). Each item tagged by a user is inserted to the Bloom filter to form the profile digest. Such a Bloom filter allows checking whether a user has tagged a given item. The profile digest can also be a Bloom filter of tags (*TagBloom*). Each tag used by a user is inserted to the Bloom filter to form the profile digest. Such a Bloom filter allows checking whether a user has used a given tag. Finally, the profile digest can be a composition of a Bloom filter of items and a Bloom filter of tags (*ItemTagBloom*). Such a Bloom filter can be used to assess if a user has used either a tag or an item. Note that the association between an item and a tag is not reflected in the digest. More specifically, the presence of the item i and the tag t in a user's profile digest does not necessarily mean that i was tagged with t.

The use of profile digest to compute an upper bound of the similarity between users first enables to limit the unnecessary exchanges of profiles during the personal network maintenance. Secondly, this enables to limit the number of users that should be contacted at query time. In P4Q, we choose to use all the available information about items and tags and implement the *ItemTagBloom* scheme to encode the digest of a profile. As we show in the experimental evaluation, this represents the most accurate solution to assess the similarity between users. We detail these issues in the following.

2.2. Bimodal Gossiping

P4Q relies on a two-mode gossip protocol as represented in Figure 1. The *lazy mode* runs *periodically* at a low frequency and is responsible for maintaining the personal network and the random view. The *eager mode* runs *on-demand* and is in charge of the collaborative query processing while refreshing a specific portion of users' personal networks. The eager mode is only activated upon queries and stops when the query is accurately computed. Queries are gossiped among the neighbors in personal networks for collecting the profiles of similar neighbors required to compute the query but which are not stored by the querier.

In short, a generic peer-to-peer gossip protocol proceeds as follows: each peer p knows a set of other peers (i.e., contact information like IP and port), called p's *view*. Periodically, p selects one peer q from its view and sends some information to q. In return, q also sends some information to p. Then p and q process the received information according to the specific application. The gossip period is referred as a *cycle*. Such protocols have been successfully used for overlay topology maintenance [Jelasity et al. 2007; Voulgaris and van Steen 2005] and information dissemination [Eugster et al. 2004]. We now describe the lazy and eager modes of the P4Q gossip protocol.

2.2.1. Maintaining Personal Networks: The Lazy Mode.

The personal network of each user is discovered and maintained through a two-layer gossip. The bottom layer, also known as the random peer sampling protocol [Jelasity et al. 2007], maintains the random view of a user: at each cycle, a user u_x sends the r digests[4] in her random view to a neighbor v_y picked uniformly at random from that view and receives r digests from v_y. Then r digests among the $2r$ digests are randomly selected to form the new random view of u_x. If two digests of the same user have different versions, the one with the more recent version is kept. v_y follows the same algorithm.

[4]The contact information of the corresponding users is also exchanged but omitted for the ease of presentation.

The top layer is in charge of tracking the similarity between user profiles and discovering new neighbors for the personal network. At each cycle, a gossip initiator u_x selects a neighbor u_y from her personal network to gossip with. This leverages the fact that exchanging information between similar neighbors significantly speeds up the convergence of the personal networks assuming that friends' friends may also be friends. Each neighbor in u_x's personal network has a timestamp that indicates for how many cycles she has not been gossiped with. The initial value of a neighbor's timestamp is set to 0 when she is added to the personal network. u_x selects the neighbor having the oldest timestamp to gossip with and the neighbor's timestamp is set to 0 while other neighbors increment their timestamps by 1. This guarantees that each neighbor has a comparable chance to participate in the gossip. u_x then sends a gossip message to u_y.

This message is composed of a subset of her neighbors' profiles, randomly selected from the c profiles stored in u_x's personal network. In turn, u_y sends back to u_x a subset of her neighbors' profiles. Then u_x computes $Score_{u_x}(u_z)$ for each received user u_z and $Score_{u_x}(v_w)$ for each user v_w in her random view. If v_w qualifies to be incorporated in the personal network, the profile of v_w is obtained by directly contacting v_w. We detail that operation shortly. User u_x (u_y) keeps in her personal network the s users with the highest (strictly positive) scores and the profiles of the top ranked c users are locally stored. Again, if u_x is aware of two different versions of user u_z's profile digest (profile), the one with the more recent version is kept in her personal network.

The bottom layer and the top layer run in parallel, that is, at each cycle, a user gossips with a neighbor from her random view and a neighbor from her personal network respectively. This ensures the network to be connected: using solely personal networks could lead to a partition if user groups exhibit completely disjoint interests. Moreover, maintaining the random view provides a chance to find new neighbors that have not been recognized by current neighbors and accelerates the personal network maintenance.

To avoid overloading the system, the transmission of profiles in the top-layer gossip follows a 3-step protocol. Algorithm 1 depicts the data exchange procedure.

—*Score estimation*: The first exchange (1-18) of the digests enables to estimate the similarity between users using the profile digests. Based on the information in the profile digest, an upper bound of similarity score between a pair of users can be computed. It is possible for a user u_z to become a neighbor of user u_x only if: (i) this estimated upper bound is larger than the similarity score between u_x and the least similar neighbor in her personal network, or (ii) this upper bound is larger than 0 while user u_x has not yet the desired number of neighbors in her personal network. Otherwise, there is no need to exchange the profile since u_z cannot qualify for the personal network of u_x.
—*Exact score computation*: The second exchange (19-27) of tagging actions that contribute to the upper-bound estimation enables to precompute the exact similarity score of each user. As only the c users' profiles having the highest scores will be stored, there is no need to transmit other neighbors' profiles.
—*Profile exchange*: The last exchange (28-33) only happens if there are indeed profiles that should be stored, namely if u_z is one the c most similar users for u_x.

We now detail how the upper bound of similarity score is estimated based on the *ItemTagBloom* digest, and how tagging actions are transmitted in the second step to compute the exact similarity score. Both a Bloom filter of items and a Bloom filter of tags are used to form a user's profile digest. When user u_x receives the profile digest of user u_z, she first queries the items she has tagged to detect their common items. Then for each of the common items, she continues querying if the tags used by herself

ALGORITHM 1: Gossiping profiles in lazy mode

```
1.  Input: $Profile(u_x)$ & received profile digests
2.  Output: new $Network(u_x)$
3.  for each received $Digest(u_z)$ do
4.    if $u_z \in Network(u_x)$ then
5.      if $Digest(u_z)$ does not change (same $Version(u_z)$) then
6.        drop $Digest(u_z)$
7.      else
8.        estimate the similarity upper-bound $UbScore_{u_x}(u_z)$
9.        if $|Network(u_x)| < s$ and $UbScore_{u_x}(u_z) > 0$ then
10.         add $u_z$ to $Candidates$
11.       else if $|Network(u_x)| = s$ then
12.         if $UbScore_{u_x}(u_z) > Min\{Score_{u_x}(u_w), u_w \in Network(u_x)\}$ then
13.           add $u_z$ to $Candidates$
14.         end if
15.       end if
16.     end if
17.   end if
18. end for
19. if $Candidates$ is not empty then
20.   for each $u_z$ in $Candidates$ do
21.     send $\langle i, t\rangle$ pairs for computing $UbScore_{u_o}(u_z)$ and receive $Score_{u_x}(u_z)$
22.   end for
23.   for each $u_z$ in $Candidates$ do
24.     if $Score_{u_x}(u_z)$ is one of the $s$ highest scores then
25.       add $u_z$ to $Network(u_x)$ with $Score_{u_x}(u_z)$ and $Digest(u_z)$
26.     end if
27.   end for
28.   for each $u_z$ added to $Network(u_x)$ do
29.     if $Score_{u_x}(u_z)$ is one of the $c$ highest scores then
30.       require $Profile(u_z)$
31.     end if
32.   end for
33. end if
```

to tag this item are also used by u_z. The score upper bound can thus be estimated by counting her tagging actions where both the concerned items and tags are used by u_z. More formally, the score upper bound can be expressed as

$$UbScore_{u_x}(u_z) = |\{\langle i, t\rangle | Tagged_{u_x}(i, t, *) \wedge Tagged_{u_z}(i, *, *) \wedge Tagged_{u_z}(*, t, *)\}|,$$

where $Tagged_{u_z}(i, *, *)$ represents an item tagged by the user u_z regardless of the tags used and $Tagged_{u_z}(*, t, *)$ represents a tag used by u_z in her profile regardless of the tagged item. Once u_z appears to be a candidate neighbor of u_x, u_x sends to her gossip destination u_y the $\langle i, t\rangle$ pairs for computing $UbScore_{u_x}(u_z)$. Then u_y counts the number of the $\langle i, t\rangle$ pairs that also appear in u_z's profile and returns the exact similarity score to u_x.

Specifically, as shown in Figure 2, when user u_1 receives the profile digest of user u_3, she first detects their common items i_1 and i_3 by querying the items tagged by herself (i_1, i_2 and i_3) in the item part of $Digest(u_3)$ ($ItemBloom$). Then u_1 queries her tags t_1, t_2, and t_3 of i_1 in the tag part of $Digest(u_3)$ ($TagBloom$) to check whether they are also used by u_3. Similarly, u_1 also queries her tag t_2 of i_3 in $Digest(u_3)$ to detect its presence. Finally, u_1 obtains the score upper bound 4. Then the 4 pairs contributing to the estimation of the score upper bound (i.e., $\langle i_1, t_1\rangle$, $\langle i_1, t_2\rangle$, $\langle i_1, t_3\rangle$ and $\langle i_3, t_2\rangle$) are sent

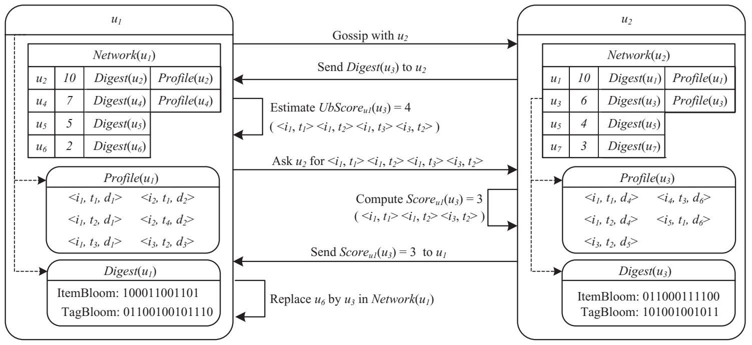

u_1
u_2
Gossip with u_2
Send $Digest(u_3)$ to u_2
Estimate $UbScore_{u1}(u_3) = 4$
(<i_1, t_1> <i_1, t_2> <i_1, t_3> <i_3, t_2>)
Ask u_2 for <i_1, t_1> <i_1, t_2> <i_1, t_3> <i_3, t_2>
Compute $Score_{u1}(u_3) = 3$
(<i_1, t_1> <i_1, t_2> <i_3, t_2>)
Send $Score_{u1}(u_3) = 3$ to u_1
Replace u_6 by u_3 in $Network(u_1)$
$Network(u_1)$
u_2 10 $Digest(u_2)$ $Profile(u_2)$
u_4 7 $Digest(u_4)$ $Profile(u_4)$
u_5 5 $Digest(u_5)$
u_6 2 $Digest(u_6)$
$Profile(u_1)$
<i_1, t_1, d_1> <i_2, t_1, d_2>
<i_1, t_2, d_1> <i_2, t_4, d_2>
<i_1, t_3, d_1> <i_3, t_2, d_3>
$Digest(u_1)$
ItemBloom: 100011001101
TagBloom: 01100100101110
$Network(u_2)$
u_1 10 $Digest(u_1)$ $Profile(u_1)$
u_3 6 $Digest(u_3)$ $Profile(u_3)$
u_5 4 $Digest(u_5)$
u_7 3 $Digest(u_7)$
$Profile(u_3)$
<i_1, t_1, d_4> <i_4, t_3, d_6>
<i_1, t_2, d_4> <i_5, t_1, d_6>
<i_3, t_2, d_5>
$Digest(u_3)$
ItemBloom: 011000111100
TagBloom: 101001001011

Fig. 2. Computing similarity scores in gossip.

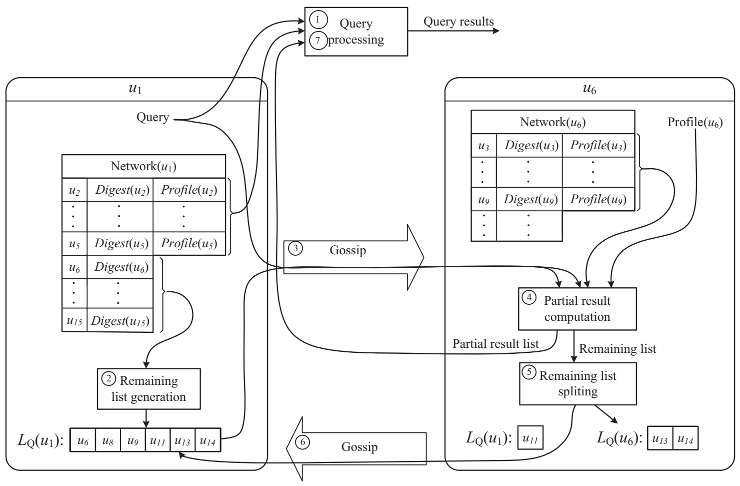
to u_2. Although both i_1 and t_3 appear in u_3's profile as indicated by her profile digest, t_3 is actually not used by u_3 to tag i_1. As a result, u_2 sends the exact similarity score 3 to u_1. If the lowest similarity score in u_1's current personal network is 2 with u_6, u_3 is added to u_1's personal network by replacing u_6.

The size of each Bloom filter is adaptively generated to guarantee a fixed false positive rate, which is dependent on the size of Bloom filter and the number of elements to insert. Any change in the profile requires regeneration of the corresponding profile digest. Note that a false positive would not lead to any incorrectness while selecting the neighbors for the personal network since a misjudgment only overestimates the score upper bound. The exact similarity score between two users is further verified in the second step of the gossip of profiles. After this step, the neighbor selection is based on the exact scores and is thus correct. The only impact of false positive is to transmit some useless tagging actions for computing the exact scores. Yet, a low false positive rate would keep this impact marginal.

Finally, the *lazy mode* runs at a low frequency keeping a low-level network traffic.

2.2.2. Processing Queries: The Eager Mode. Should a user be able to compute a query based on all the profiles of her personal network, the result would be exact (recall of 1). However, for space and result freshness reasons, only the profiles of the c neighbors having the highest scores are locally stored. This can be used to compute a partial result to the query. Yet, the user has to contact other users in the network for collecting the missing profiles. This is achieved in a collaborative and distributed manner by gossiping the queries in the personal network using the top-layer protocol in eager mode. The queries are gradually processed collaboratively by the querier and other users reached by the queries. The reason for only gossiping the queries within personal networks is twofold. First, it is unlikely that the profiles stored by random neighbors are required by the querier. Second, applying various gossip frequencies (generated by the on-demand nature of the eager mode) at the bottom layer may jeopardize the uniform randomness of the underlying network topology [Jelasity et al. 2007].

The eager mode of P4Q works as follows and the first cycle of gossip is illustrated in Figure 3. The querier u_x first processes her query Q based on the profiles in her personal network (①). This provides a partial and local result to the query. The *remaining list* of user u_x for query Q, denoted $L_Q(u_x)$, is the set of users from her personal network whose profiles are not stored locally but would contribute to the query processing. While the query is collaboratively answered by potentially all the neighbors in u_x's personal network, only the neighbors who have used the tags in the query are involved in the query processing. The remaining list of user u_x for her query Q is thus composed of users in her personal network who have used at least one tag in the query but are



1 Query 7 processing
Query results
u_1
Query
Network(u_1)
u_2 Digest(u_2) Profile(u_2)
u_5 Digest(u_5) Profile(u_5)
u_6 Digest(u_6)
u_{15} Digest(u_{15})
2 Remaining list generation
$L_Q(u_1)$: u_6 u_8 u_9 u_{11} u_{13} u_{14}
3 Gossip
u_6
Network(u_6)
Profile(u_6)
u_3 Digest(u_3) Profile(u_3)
u_9 Digest(u_9) Profile(u_9)
4 Partial result computation
Partial result list
Remaining list
5 Remaining list spliting
6 Gossip
$L_Q(u_1)$: u_{11}
$L_Q(u_6)$: u_{13} u_{14}

Fig. 3. Query processing in eager mode (1^{st} cycle).

not locally stored by u_x (②). This information is clearly reflected in their profile digests since the tags used by each of the neighbors is present in their *ItemTagBloom* digests. Note that the remaining list, being built using the Bloom filters, might contain some users who did not use the query tags due to the false positive answers. Yet, this has no impact on the result quality as no useful profiles would be missing. As we will see, the profiles that cannot contribute to the query processing but are added to the remaining list do not interfere and are automatically filtered out when the query is processed. The remaining list also contains, for each user, the desired version of her profile, indicated by the corresponding version of her profile digest in the querier's personal network when the query is issued. We use T_Q in the following to denote the time when the query is issued.

Those profiles of users in the remaining list are discovered through gossip. User u_x initiates a gossip with the neighbor u_y having the oldest timestamp in $L_Q(u_x)$ and attaches the query (as well as the time when it is issued (T_Q)), and the remaining list to the gossip message containing the profile digests she wants to send to u_y as described in the lazy mode (③). When u_y receives the message, she checks whether she locally stores the profiles of the users in $L_Q(u_x)$, removes them from the remaining list, and processes her share of the query locally (④). More specifically, if user u_z's profile stored by u_y is more recent than or as recent as its version indicated in the remaining list, u_y processes the query with the tagging actions in u_z's profile, which occurred before the query is issued. This ensures the consistency of the query results, that is, the profile used in the query processing is exactly the same as expected by the querier when she issued the query. Then the remaining list is updated by removing u_z. This updated remaining list is then split into two parts (⑤): a portion α ($0 \leq \alpha \leq 1$) is sent back to the querier in her gossip message containing the profile digests (⑥), the remaining portion forms her remaining list for the query Q: $L_Q(u_y)$. The intuition is that this user will take care of a portion of the remaining list herself through gossip while the portion sent back to the querier will be processed by the querier through other of her neighbors. The partial result of the query is sent back to the querier in a message independent of the gossip. A list of users whose profiles are used for the computation is also sent to the querier in the same message. This information is used to estimate the quality of the current results. The more users' profiles have been used for the query processing,

ALGORITHM 2: Query processing at the querier

```
1.  Input: querier $u_x$'s query $Q$ & $Network(u_x)$
2.  Output: personalized query results of $Q$
3.  process Q with the profiles in $Network(u_x)$
4.  if $u_x$ stores all her neighbors profiles then
5.    display query results and return
6.  else
7.    build the remaining list $L_Q(u_x)$
8.    repeat
9.      gossip with a neighbor $u_y$ in $L_Q(u_x)$
10.     receive new $L_Q(u_x)$ from $u_y$
11.     receive partial results from collaborating users
12.     compute and display new results with available information
13.   until all neighbors' profiles are used for query processing
14. end if
```

the closer the results should be to the ideal ones. At the end of the first cycle, the querier updates the query results with the partial result received during this cycle (⑦).

In the second cycle, both u_x and u_y gossip with one of their neighbors that are also in their remaining lists if the sizes of their remaining lists are larger than 0. If none of the users in the remaining list is the neighbor of the gossip initiator, a user is chosen randomly from her remaining list as gossip destination. Contacting such users ensures to find at least one profile of interest to the querier. Receiving the gossip message, the gossip destinations of u_x and u_y do the same processing as u_y did in the first cycle. At the end of the second cycle, the querier updates the query results with the new available partial results received during this cycle.

This process continues until none of the users reached by Q has a remaining list. At this moment, the *accurate* (recall of 1) personalized results, based on the information of whole personal network, are obtained. The query results are in fact updated and displayed at the end of each cycle and the querier can estimate the quality of the results according to the number of profiles that have been used for the query processing and decide whether she is satisfied. The querier stops waiting the incoming partial result lists if all her neighbors' profiles are used for the processing.

Algorithm 2 is the query processing at the querier, whereas Algorithm 3 shows how a query is gossiped between two users. The gossip initiator is the user who forwards the query and the remaining list, and the gossip destination is the user who processes the query and splits the remaining list.

The splitting process, specified by the splitting factor α, is used to avoid taking the same profile into account several times during the query processing, if this profile is stored by more than one user reached by the query. This ensures that every user participating in the query processing is in charge of a different part of the initial remaining list and guarantees the accuracy of the final results. The optimality of α in P4Q will be discussed later.

As opposed to the lazy mode, the eager mode runs at a higher frequency in order to provide quick responses for the queries. Although it temporarily increases the network traffic due to the gossip exchanges of profiles, it significantly helps updating the personal networks of the users participating in the gossip (Section 3.4.1).

2.3. Collaborative Top-k Query Processing

We illustrate in this section the collaborative query processing in P4Q in the context of top-k processing.

ALGORITHM 3: Gossiping queries in eager mode

```
1.  Gossip Initiator ($u_{init}$)
2.  for each cycle do
3.    if $|L_Q(u_{init})| > 0$ then
4.      if $\exists u_z \in L_Q(u_{init})$ & $u_z \in Network(u_{init})$ then
5.        $u_{dest} \leftarrow u_z$ with maximum timestamp
6.        set $u_{dest}$'s timestamp to 0
7.      else
8.        select $u_{dest}$ from $L_Q(u_{init})$
9.      end if
10.     send $Q$ and $L_Q(u_{init})$ to $u_{dest}$ in gossip message
11.     receive gossip message containing new $L_Q(u_{init})$ from $u_{dest}$
12.     maintain personal network as in lazy mode
13.   end if
14. end for

15. Gossip Destination ($u_{dest}$)
16. loop
17.   receive gossip message containing $Q$ and $L_Q(u_{init})$ from $u_{init}$
18.   for each $u_z$ in $L_Q(u_x)$ do
19.     if $Profile(u_z)$ is stored by $u_{dest}$ & $Version(u_z) \geq T_Q$ then
20.       prepare $u_z$ for query processing and remove $u_z$ from $L_Q(u_{init})$
21.     end if
22.   end for
23.   $L_Q(u_{dest}) \leftarrow (1-\alpha)*|L_Q(u_{init})|$ random users from $L_Q(u_{init})$
24.   $L_Q(u_{init}) \leftarrow L_Q(u_{init}) \backslash L_Q(u_{dest})$
25.   send $L_Q(u_{init})$ to $u_{init}$ in gossip message
26.   process $Q$ with each prepared profile $u_z$ with $Tagged_{u_z}(i,t,d)$ and $d \leq T_Q$
27.   send partial result to the querier
28.   maintain personal network as in lazy mode
29. end loop
```

Queries and Scoring. We consider a query $Q = \{t_1, \ldots, t_n\}$, issued by user u_x with a set of tags $t_1, \ldots, t_n$ at time T_Q. The personalized top-k processing for Q aims to return the k items having the highest relevance scores from u_x's personal network. More specifically, we define the score of item i for user u_y and query Q as the number of tags in Q used by u_y to annotate i, that is, $Score_{u_y,Q}(i) = |\{t \mid t \in Q, d \leq T_Q, Tagged_{u_y}(i,t,d)\}|$. We define the relevance score of item i for user u_x's query Q as the sum of $Score_{u_y,Q}(i)$ of each neighbor u_y in u_x's personal network, that is,

$$Score(Q,i) = \sum_{u_y \in Network(u_x)} Score_{u_y,Q}(i).$$

Alternative monotonic scoring functions can also be used to compute such user-specific relevance score.

Top-k Processing in P4Q. As presented before, in P4Q, a query is processed in collaboration among the querier and the users reached by the query. We here describe how the partial results are computed by each user and how the querier updates the top-k results upon receiving new partial results at each cycle.

In P4Q, once user u_y receives query Q, she computes a partial result for Q with the parts (not more recent than the query time) of the profiles she stores that should be used for the query processing. These profiles can be either her own profile or those stored in her personal network. We denote this set of profiles $GoodProfiles(u_y, Q)$. u_y

ALGORITHM 4: Per cycle top-k processing of the querier

```
1. Input: $u_x$'s query $Q$ & candidate heap & old partial result lists & new partial result lists
2. Output: new top-$k$ items
3. $ScanningLists \leftarrow$ new partial result lists
4. $ScanningPosition \leftarrow 1$
5. while worst-case score of the $k^{th}$ item in candidate heap < max{best-case scores of items in
   candidate heap but not in top-$k$} do
6.   for each partial result list $l$ in $ScanningLists$ do
7.     get item $i$ in the $ScanningPosition$ of $l$
8.     update the last seen value and last scanned position of $l$
9.     if $i \in$ candidate heap then
10.      update $i$'s best-case score and worst-case score
11.    else
12.      add $i$ in candidate heap
13.    end if
14.    update the best-case scores for items in candidate heap
15.    re-order candidate heap
16.  end for
17.  $ScanningPosition \leftarrow ScanningPosition + 1$
18.  for each partial result list $l \in$ old partial result lists do
19.    if last scanned position = $ScanningPosition - 1$ then
20.      add $l$ to $ScanningLists$
21.    end if
22.  end for
23. end while
```

computes a partial relevance score for each item appearing in these profiles. With respect to the definition of the overall relevance score $Score(Q, i)$, the partial relevance score of an item i can be computed as the sum of $Score_{u_z,Q}(i)$ for each $Profile(u_z)$ in $GoodProfiles(u_y, Q)$, that is,

$$PartialScore_{u_y}(i) = \sum_{Profile(u_z) \in GoodProfiles(u_y,Q)} Score_{u_z,Q}(i).$$

The partial result for the query Q is a list containing all the items having positive partial relevant scores and the items are ranked in descending order of their scores.

The querier's local processing before gossiping the query is also carried out this way and the k items ranked on top of the resulting list are displayed as the first query results for the querier.

Existing top-k techniques cannot be directly used within P4Q as the partial result lists in P4Q are computed on-the-fly and asynchronously provided to the querier. So we adapt the classical NRA (No Random Access) [Fagin 2002] algorithm to P4Q while minimizing the processing time. In P4Q, at the end of each cycle, k items are returned to the querier. Algorithm 4 shows the pseudocode of the top-k processing at a given cycle.

For any query, at a given cycle, the querier already has the partial result lists used for the top-k processing in the previous cycle and the resulting candidate heap of items, where each item has a best-case score and a worst-case score and they are ranked according to their worst-case scores as in classical NRA. In NRA, the ranked lists are scanned sequentially in parallel. The worst-case score takes the most pessimistic assumption that if an item has not been seen in some lists while scanning, then it does not exist in those lists. Alternatively the best-case score takes the most optimistic assumption that its scores in those lists are equal to the scores of the last seen items in those lists. In the current cycle, the querier receives some new partial result lists. The query is processed using all the available information to compute the new top-k items.

The processing begins by scanning the new partial result lists sequentially in parallel. For each partial result list (old or new), the last scanned position is maintained. Each time the cursor reaches a new position, all the partial result lists stopped at this position before should continue to be scanned with the currently scanned ones. At this point, we guarantee that each partial result list is scanned only once during the whole processing. Once a new item is encountered in a partial result list, the querier first checks if it is already in the candidate heap. If it exists, its best-case score and worst-case score are updated. Otherwise, it is added to the heap. The best-case scores of other items in the candidate heap should be accordingly updated. The scores are computed using the same assumption as in NRA. The candidate heap is kept sorted in descending order of the worst-case scores. For the items with equal worst-case scores, the ones with larger best-case scores are ranked ahead. The processing stops when none of the items out of the first k items has a best-case score larger than the worst-case score of the k^{th} item. Several optimizations are possible to incorporate into the basic algorithm, like not reranking the candidate heap once an item is modified, but they are out of the scope of this article.

2.4. Analysis of the Query Processing

To analyze the efficiency of the query processing, we consider a simplified model. We assume that each time a query is gossiped, the same number of profiles, noted as X, can be found in the gossip destination's local storage. As guaranteed by our gossip strategy described in Section 2.2.2, at least 1 profile can be found ($X \geq 1$) by contacting its owner in the gossip.

THEOREM 2.1. *Given a query Q and the querier u_x's remaining list of length L, u_x gets the best results that her personal network can provide within $R(\alpha)$ cycles, where*

$$R(\alpha) = \begin{cases} 1 - \log_{\alpha}[(1-\alpha)L/X + \alpha] & 0.5 \leq \alpha < 1 \\ 1 - \log_{1-\alpha}[\alpha L/X + (1-\alpha)] & 0 < \alpha < 0.5 \\ L/X & \alpha = 0, \alpha = 1 \end{cases} .$$

PROOF. As described in the algorithm, at a given cycle, a user with her remaining list of length l for the query Q initiates a gossip with one of her neighbors. After X profiles are found, the length of her remaining list becomes $\alpha(l - X)$ while her neighbor obtains a remaining list of length $(1-\alpha)(l-X)$. If $0.5 \leq \alpha \leq 1$, comparing to her neighbor, the gossip initiator has a longer (equal) remaining list. Meanwhile, among all gossip initiators in this cycle, the one possessing the longest remaining list before should still have the longest after this cycle. So at the end of each cycle, it is always the user u_x who has the longest remaining list as she first gossips the query Q. From the definition, we know that u_x gets the best results that her personal network can provide when none of the users reached by Q has a remaining list, that is, the length of u_x's remaining list becomes 0 as she has the longest one. Note the length of u_x's remaining list after the r^{th} cycle as $L(r)$, we have

$$\begin{aligned} L(1) &= \alpha(L - X), \\ L(2) &= \alpha[L(1) - X] = \alpha^2 L - \alpha^2 X - \alpha X, \\ &\dots \end{aligned}$$

$$\begin{aligned}
L(r) &= \alpha[L(r-1) - X] = \alpha^r L - \alpha^r X - \alpha^{r-1}X - \cdots - \alpha X \\
&= \alpha^r L - \sum_{i=1}^{r} \alpha^i X \\
&= \begin{cases} \alpha^r L - \frac{\alpha(1-\alpha^r)}{1-\alpha} X & 0.5 \le \alpha < 1 \\ L - rX & \alpha = 1 \end{cases}.
\end{aligned}$$

For u_x to get the best results in $R(\alpha)$ cycles, it is sufficient to let $L[R(\alpha)] = 0$, then we can get

$$R(\alpha) = \begin{cases} 1 - \log_\alpha[(1-\alpha)L/X + \alpha] & 0.5 \le \alpha < 1 \\ L/X & \alpha = 1 \end{cases}.$$

If $0 \le \alpha < 0.5$, similarly, we can obtain the length of the longest remaining list after the rth cycle as

$$\begin{aligned}
L(r) &= (1-\alpha)[L(r-1) - X] = (1-\alpha)^r L - \sum_{i=1}^{r}(1-\alpha)^i X \\
&= \begin{cases} (1-\alpha)^r L - \frac{(1-\alpha)[1-(1-\alpha)^r]}{\alpha} X & 0 < \alpha < 0.5 \\ L - rX & \alpha = 0 \end{cases}.
\end{aligned}$$

Hence, for the longest remaining list to become 0, we have

$$R(\alpha) = \begin{cases} 1 - \log_{1-\alpha}[\alpha L/X + (1-\alpha)] & 0 < \alpha < 0.5 \\ L/X & \alpha = 0 \end{cases}. \qquad \square$$

THEOREM 2.2. *Given L and X, the number of cycles for the querier u_x to get the best results for her query Q, $R(\alpha)$, monotonically increases with α if $0.5 \le \alpha < 1$ and monotonically decreases with α if $0 < \alpha < 0.5$. The minimum number can be achieved at $\alpha = 0.5$.*

PROOF. Let $0.5 < \alpha_2 < \alpha_1 < 1$, we have

$$\begin{aligned}
&R(\alpha_1) - R(\alpha_2) \\
&= (1 - \log_{\alpha_1}[(1-\alpha_1)L/X + \alpha_1]) - (1 - \log_{\alpha_2}[(1-\alpha_2)L/X + \alpha_2]) \\
&= \frac{\ln[(1-\alpha_2)L/X + \alpha_2]}{\ln \alpha_2} - \frac{\ln[(1-\alpha_1)L/X + \alpha_1]}{\ln \alpha_1} \\
&= \frac{\ln[(1-\alpha_2)L/X + \alpha_2]\ln \alpha_1 - \ln[(1-\alpha_1)L/X + \alpha_1]\ln \alpha_2}{\ln \alpha_1 \ln \alpha_2}.
\end{aligned}$$

Considering $L \ge X$ and $\alpha_2 < \alpha_1$, we have

$$\begin{aligned}
&[(1-\alpha_2)L/X + \alpha_2] - [(1-\alpha_1)L/X + \alpha_1] \\
&= (\alpha_1 - \alpha_2)(L/X - 1) > 0.
\end{aligned}$$

Then $\ln[(1-\alpha_2)L/X + \alpha_2] > \ln[(1-\alpha_1)L/X + \alpha_1]$.
Moreover, as $\ln \alpha_2 < \ln \alpha_1 < 0$, we have

$$R(\alpha_1) - R(\alpha_2) > 0.$$

Hence, $R(\alpha)$ monotonically increases with α if $0.5 \le \alpha < 1$.
Similarly, for $0 < \alpha_2 < \alpha_1 < 0.5$, let $\beta_1 = 1 - \alpha_1$ and $\beta_2 = 1 - \alpha_2$, we have $0.5 < \beta_1 < \beta_2 < 1$. Then $R(\alpha_1) - R(\alpha_2) = R(\beta_1) - R(\beta_2) < 0$.

Hence, $R(\alpha)$ monotonically decreases with α if $0 < \alpha < 0.5$. Moreover,

$$\begin{aligned} R(0.5) - R(1) &= R(0.5) - R(0) \\ &= 1 - log_{0.5}(0.5L/X + 0.5) - L/X \\ &= log_{0.5}\frac{2^{L/X}}{L/X + 1} < 0. \end{aligned}$$

Therefore, $R(\alpha)$ gets the minimum number at $\alpha = 0.5$. □

THEOREM 2.3. *The number of users involved in the processing of a query Q is bounded by $2^{R(\alpha)}$. The number of partial results sent to the querier for her query Q is bounded by $2^{R(\alpha)} - 1$.*

PROOF. Suppose all the users involved in the query processing finish their tasks simultaneously, that is, the sizes of their remaining lists reach 0 at the same cycle. As we know, at the 1^{st} cycle, one new user is involved except for the querier. Hence the total number of involved users is 2 ($= 2^1$). Using mathematical induction, if at the r^{th} cycle, 2^r users are involved and none of them has finished their remaining lists, then at the $(r + 1)^{th}$ cycle, each of them gossip with another user, which implies that 2^r new users are involved. As a result, the total number of users is 2^r+2^r=2^{r+1}. Actually, at a given cycle, the size of the remaining list is different for each user if $\alpha \neq 0.5$. The users having no remaining list would stop the eager gossip so that no more new users would be further involved by them. Therefore $2^{R(\alpha)}$ is an upper bound of the number.

If at least one profile is found among profiles stored by each involved user and these profiles have at least one item tagged by a tag in the query Q before it is issued, each user should send her partial result to the querier. This implies the upper bound is $2^{R(\alpha)} - 1$ because the querier has her partial result locally. □

THEOREM 2.4. *The number of eager gossip messages for transmitting the remaining lists during the processing of a query Q is bounded by $2 \times (2^{R(\alpha)} - 1)$.*

PROOF. We begin by counting the number of gossips occurred during the processing of Q. At the 1^{st} cycle, one gossip is done between the querier and one of her neighbors. Assuming all the users involved in the processing finish their remaining lists at the same time, at the 2^{nd} cycle both the querier and her neighbor gossip with another user, implying 2 ($= 2^{2-1}$) gossips. This process continues until no user has a remaining list. In fact, at the r^{th} cycle, 2^{r-1} gossips are done. Therefore the total number of gossips during the first r cycles is $\sum_{i=1}^{r} 2^{r-1} = 2^r - 1$. During each cycle of eager gossip, 2 messages are exchanged for the transmission of the remaining lists: one for forwarding the gossip initiator's remaining list and one for returning her the new remaining list. Hence, the total number of the eager gossip messages is $2 \times (2^{R(\alpha)} - 1)$ if the processing ends at cycle $R(\alpha)$. Again, this number can be achieved only when $\alpha = 0.5$ and it is in fact an upper bound. □

2.5. Coping with Profile Dynamics

Users in collaborative tagging systems are usually active and continuously tagging new items in the systems. This results in profile changes which in turn potentially impact the similarity between users. Personal networks should be updated to reflect such changes. When a user changes her profile, all the replicas of her profile should be updated in order to take the new information into account while refining the personal network. As this profile may be later used by other users to process their queries, timely update may directly affect the accuracy of the resulting top-k items.

In this section, we investigate how the gossip protocol in both lazy and eager modes can be fine-tuned respectively to cope with profile dynamics. In P4Q, this is achieved collaboratively by users participating in the gossip with two basic operations: *self-promotion* and *mutual-aid*.

Self-Promotion. Self-promotion consists for a user in proactively disseminating her profile upon changes. In the initial algorithm (i.e., P3Q [Bai et al. 2010]), u_x picks u_y with the oldest timestamp to gossip with in her personal network. The neighbor relation in P4Q is not necessarily symmetric and u_y may not consider u_x as her neighbor. Therefore the changes in u_x's profile might not be immediately taken into account, until u_x encounters a similar neighbor, who also considers u_x as a neighbor in her personal network.

To address this issue, when u_x changes her profile, she proactively promotes her new profile by picking from her personal network the most similar user u_z instead of the user u_y with the oldest timestamp to gossip with. The intuition is that even if the neighbor relation is not symmetric, as u_z has the highest similarity score, she is also the most likely user, among the neighbors of u_x, to store u_x's profile. Gossiping with u_z would first help u_z to update u_x's profile with high probability. More importantly, in the following cycles, u_z can further gossip u_x's profile to her own neighbors if she indeed stores that profile in her personal network. Self-promotion helps disseminating the new profile, starting with the users who are the most likely to be interested in u_x's new profile. Therefore, self-promotion speeds up the propagation of profile changes over the network.

Self-promotion is only allowed in the first cycle after a user changes her profile. Then the user continues with the standard gossip algorithm selecting the destination based solely on the timestamp. This is to avoid that too active users, who frequently change their profiles, keep gossiping with a handful of users, which are the most similar, preventing other users who are interested in their profiles from receiving the new information.

It is worth noticing that self-promotion is only applied in lazy mode. Effectively, even if the eager mode usually generates a wave of refreshment in the profiles, the main goal of the eager mode is to solve a query. As described in Section 2.2.2, the gossip destination in eager mode should be preferentially selected from the remaining list to guarantee the most efficient query processing.

Mutual-Aid. In P4Q, upon gossip for personal network maintenance (top layer of the lazy mode and the eager mode), the gossip initiator sends a subset of the profiles stored in her personal network to the gossip destination which in turn does the same. This subset is composed of a random subset of the similar profiles during standard operation. When a user's profile is updated, propagating updates may take some time. Mutual-aid consists in having each user gossip an update when inconsistencies are detected. Typically, if a user receives an out-of-date profile from her gossip initiator, that is, the version of the received profile digest is older than that of the one in the her personal network, she will gossip the up-to-date version of that profile regardless of the subset selection described in Section 2.2. More specifically, the gossip message is preferentially composed of the profiles which have been changed but not noticed by the gossip initiator. If the number of such profiles is smaller than the desired size of the gossip message, the missing profiles are randomly selected from the similar profiles as in the standard operation.

Since personal networks of similar users are likely to overlap when the system has converged to a relatively stable state, mutual-aid enables to efficiently update profiles within groups of similar users through further gossiping.

Note that the mutual-aid between gossip initiator and gossip destination can be applied in both lazy mode and eager mode. Regardless of the user selected as gossip destination, she can always propose new versions of profiles to the gossip initiator through gossiping should she have them.

3. EXPERIMENTAL EVALUATION

In this section, we report on the evaluation of P4Q. We first describe in Section 3.1 the delicious dataset, the different scenarios, and the profile digests used for the evaluation. In Section 3.2, we assess the efficiency of the lazy mode for the personal network maintenance, and that of the eager mode for the top-*k* processing. We then focus on the cost of P4Q with respect to storage and bandwidth consumption in Section 3.3. The evaluation shows that P4Q provides accurate top-*k* results in a short period of time and with limited storage and bandwidth. We highlight how users can adjust the number of profiles they locally store according to the latency or the accuracy of their queries. We also evaluate in Section 3.4 the ability of P4Q to deal with profile changes and user departures. We finally report on the scalability of P4Q as well as its generalization with respect to alternative similarity measures in Section 3.5, and discuss its feasibility in real systems in Section 3.6.

3.1. Experimental Setup

3.1.1. Dataset and Query Generation. The evaluation of P4Q has been conducted using PeerSim [Jelasity et al. 2004; Montresor and Jelasity 2009], an open-source simulator for P2P protocols. The dataset used in the evaluation was crawled in January 2009 from delicious. The crawl started from a random seed URL tagged by hundreds of users. Then the tagging actions of these users were crawled. After that, the tagging actions of the users who have tagged at least one URL involved in the previously crawled tagging actions were further crawled. We iterated on this process to enlarge our crawl. The dataset contains 13,521 distinct users who participated in 31,833,700 tagging actions, involving 4,741,631 distinct items and 620,340 distinct tags. Note that only the users whose tagging actions were fully crawled are considered. The distribution of tagging behaviors follows a long-tail distribution, due to the fact that most items and tags are used by few users [Mislove et al. 2007]. We reduce the dataset by randomly picking 10,000 users and restricting their profiles to the items and tags used by at least 10 distinct users. Note that this does not affect the top-*k* results as only the items ranked at the tail of the candidate list are removed from the dataset. Those items are hardly involved in the final results.

The remaining dataset contains 101,144 items, 31,899 tags, and 9,536,635 tagging actions.[5] In the experiments reported shortly, each user processes exactly one query: one item was randomly picked from the user's profile, the query of that user was then generated with the tags used by that user to annotate this item following the assumption that the tags used by a user to tag an item are precisely those she would use to search for that particular item.

3.1.2. System Setting. As defined in Section 2.1, each user maintains the *s* users having the highest similarity scores with her in her personal network, and stores *c* profiles. To guarantee that the top-*k* items for a query are derived from a search space containing sufficient choices, the size of personal network *s* is set to 1,000 in our simulations. In fact, regardless of the size of personal network, the querier can get the accurate results

[5]Interestingly, although there are only about 3,000 most frequent English words, the cleaned dataset contains ten times that number of tags. This is due to the multiword expression, like *socialnetwork*, *socialsearch*, *socialresponsiblility*, etc., which gives a more precise description of the items. This also gives a hint of the ability of the tagging vocabulary to grow infinitely.

Table I. Distribution of Stored Profiles (c)

c	10	20	50	100	200	500	1,000
$\lambda = 1$	36.79%	36.79%	18.39%	6.13%	1.53%	0.31%	0.06%
$\lambda = 4$	2.06%	8.25%	16.49%	21.99%	21.99%	17.59%	11.73%

within a limited number of cycles (Theorem 2.1). As mentioned earlier, each user stores c profiles of the most similar neighbors in her personal network. Several values for c are considered in the evaluation. The goal of P4Q is to provide users with an adaptive system where they can trade the number of profiles to store in their personal networks and their activities in the system depending on their requirements with respect to the query results and their capability in both storage and bandwidth.

To emphasize the effectiveness of our protocol, we first consider uniform systems where all users have identical storage capacity. We vary the value of c and it is set to 10, 20, 50, 100, 200, 500, or 1,000 respectively in 7 different scenarios. Two heterogeneous settings with respect to storage capability are then considered, following a Poisson distribution (the parameter λ of the Poisson distribution is set to 1 and 4 respectively). The detailed distribution is depicted in Table I. In the $\lambda = 1$ scenario, more than 73% users only store 10 or 20 profiles. This can be considered as a network where the users are, for instance, mobile phones with limited memory. In contrast, the $\lambda = 4$ scenario mimics a network where the majority of users can provide significant space to store their personal networks.

3.1.3. Profile Digest Selection. As described in Section 2.2.1, P4Q relies on the digests of profiles containing both tags and items, encoded using the *ItemTagBloom* scheme, to estimate the similarity score upper bound during the gossip. In order to back up our choice, we evaluate this scheme against two natural alternatives where the users rely only on the items (*ItemBloom*) or the tags (*TagBloom*), to estimate the score upper bound using profile digests. We also consider the third alternative that uses *ItemBloom* but does not estimate the score upper bound. In this latter case, the exact similarity is computed if there is any common item between two profile digests of *ItemBloom*. This is in fact the mechanism used in the initial algorithm, P3Q [Bai et al. 2010], to eliminate the profile transmission during a gossip. The evaluation shows that *ItemTagBloom*: (i) outperforms the third alternative (P3Q) by estimating the upper bound as it is more effective to detect the similarity between users; (ii) achieves the best balance between estimation accuracy and bandwidth (storage) requirement.

We first explain how the score upper bound can be estimated based on *ItemBloom* and *TagBloom*.

—*ItemBloom.* If the items are inserted in the Bloom filter to form the profile digest, u_x first checks if the items from her profile are encoded in u_z's digest. The score upper bound is estimated by u_x as the number of her tagging actions related to the common items with u_z, contained in u_z's digest, that is,

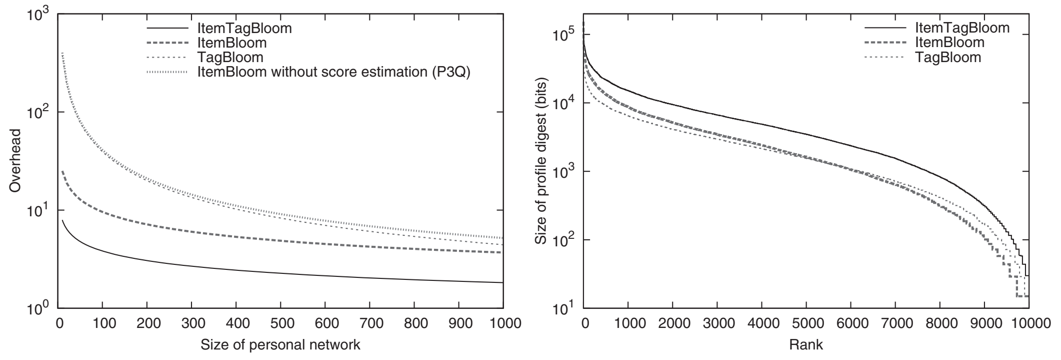
$$UbScore_{u_x}(u_z) = |\{\langle i, t\rangle | Tagged_{u_x}(i, t, *) \wedge Tagged_{u_z}(i, *, *)\}|.$$

Only the tagging actions involved in the computation of score upper bound may contribute to the exact similarity score.

—*TagBloom.* If the tags in a profile are inserted in the Bloom filter to form the profile digest, u_x first checks if the tags from her profile are encoded in u_z's digest. The score upper bound is estimated by u_x as the number of her tagging actions related to these common tags, that is,

$$UbScore_{u_x}(u_z) = |\{\langle i, t\rangle | Tagged_{u_x}(i, t, *) \wedge Tagged_{u_z}(*, t, *)\}|.$$

Similarly, only these tagging actions may contribute to the exact similarity score.



(a) accuracy of similarity upper-bound estimation (b) size of profile digests

Fig. 4. Comparison of different profile digests.

Figure 4 compares the computation overhead and the potential storage and bandwidth requirement of these approaches. The overhead is computed by dividing the minimum number of exact similarity score computations necessary to obtain a personal network of size s by the desired size s. Whether the exact similarity score of a user needs to be computed depends on the estimation of the similarity upper bound as we described in Section 2.2.1. The larger the overhead, the less accurate the estimation of the similarity upper bound.

The average computation overhead for different sizes of personal network is shown in Figure 4(a). We observe that with a false positive rate (*FP*) close to 0, the smaller the desired personal network, the more significant the saving in computation thanks to the upper-bound estimation. *ItemBloom* without score estimation, referring to the second step of the lazy gossip in the initial algorithm (P3Q [Bai et al. 2010]) always requires the most computation. With a personal network of 1,000 neighbors in our setting, *ItemTagBloom* outperforms both *ItemBloom* and *TagBloom*. The average overhead of *ItemTagBloom* accounts for only 49% and 41% of each respectively.

Figure 4(b) compares the size of the profile digest built with different Bloom filters. The digests are ranked in descending order of their sizes that guarantee a false positive rate of 0.1%. On average, the size of a profile digest in *TagBloom* is 2,689 bits and that of a profile digest in *ItemBloom* is 3,588 bits. The average size of a profile digest in *ItemTagBloom* is thus 6,277 bits, which is the sum of the aforesaid ones. Here we use an adaptive profile digest for each user's profile. In other words, the size of the Bloom filter is dependent on the number of items or tags in the user's profile. In contrast, if the profile digests of uniform size are used as in P3Q, to guarantee that 99% of *ItemBloom* have a false positive rate of 0.1%, 20,000 bits are needed for each digest.

As expected, *ItemTagBloom* provides the most accurate estimation of the similarity score upper bound but requires the largest profile digests. Yet, the size of the largest profile digest of *ItemTagBloom* in our experiments is 188,837 bits (23.6KB) and the average size is only 6,277 bits (0.78KB). We thus focus in our experiments on the profile digest in *ItemTagBloom* as it requires much less computation overhead than the other alternatives. Different profile digests might be used for other applications to achieve the desired balance between the accuracy of estimation and the size of profile digest.

Figure 5 shows the computation overhead of *ItemTagBloom* with different false positive rates. A false positive rate of 0.1% guarantees almost the same level of accuracy as there is no false positive. The only way to avoid false positive is not to use a Bloom filter but to use the list of items or tags as the profile digest instead. However, this multiplies

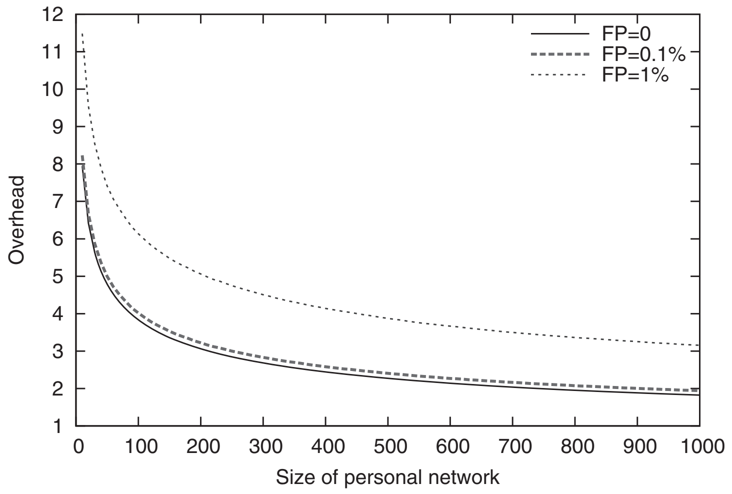


Fig. 5. Overhead of *ItemTagBloom* with different false positive rates.

the size of the digest by a large factor depending on the size of an item identifier or a tag (12.4 in our dataset). Therefore, we use a false positive rate of 0.1% in the following.

3.2. Qualitative P4Q Evaluation

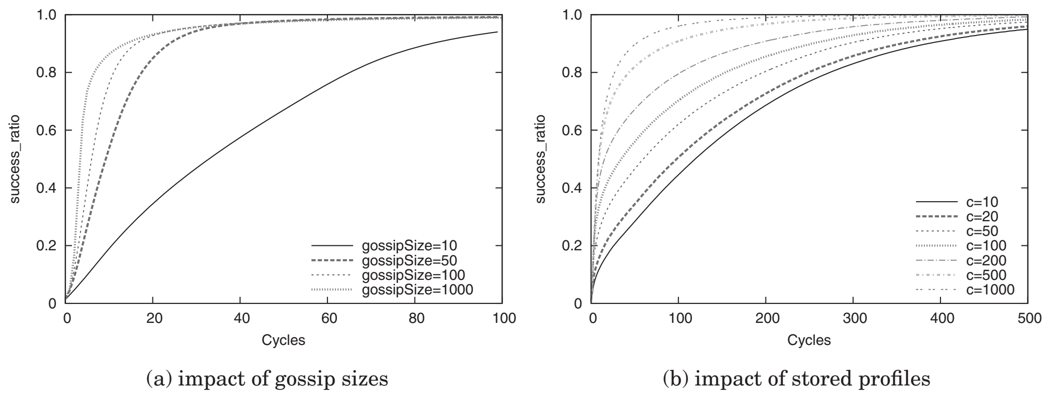
3.2.1. Personal Network Maintenance in Lazy Mode. We first evaluate the ability of P4Q to discover users having similar tagging behaviors. We assume that each user builds her personal network by first discovering the contact information of any user currently in the system using the random peer sampling protocol. The s users with the highest similarity score are gradually integrated in the personal network through the gossip protocol in lazy mode. We evaluate the convergence property of the personal networks by measuring the number of gossip cycles required for users to build their personal networks.

The quality of a user's personal network is measured as its success ratio to the ideal one obtained offline using the global information about all users' profiles. The success ratio is defined as the number of users that are in the personal network (and should be) over the total number of neighbors in the ideal personal network. The speed of convergence is then measured by the average of the resulting success ratios over all users for each cycle, that is,

$$success\_ratio = \frac{1}{|U|} \sum_{u_x \in U} \frac{\textit{Number of Good Neighbors in Current Network}}{\textit{Number of Neighbors in Ideal Personal Network}}.$$

success_ratio reaches 1 when all users converge to their ideal personal networks.

There is a trade-off between convergence speed and bandwidth consumption orchestrated by the number of profiles exchanged in gossip. The more profiles are exchanged at each cycle, the faster users discover new neighbors for their personal networks (convergence) and the more bandwidth is required. Figure 6(a) compares different number of profiles exchanged in each cycle (gossipSize) and confirms its impact on convergence. In this experiment, each user stores all the profiles in her personal network and 10 random profile digests are gossiped in the bottom layer of the protocol (lazy mode). In the following experiments, at most 50 profiles are exchanged in each cycle as it guarantees similar convergence while requiring only 23% of the peak bandwidth when all the profiles in the personal network are exchanged.



(a) impact of gossip sizes (b) impact of stored profiles

Fig. 6. Convergence of personal networks.

Figure 6(b) shows the convergence speed assuming uniform storage across users. Not surprisingly, the more profiles are stored, the faster the users successfully build their personal networks. More profiles in the personal network gives the current neighbors more opportunities to discover new neighbors, increasing the diversity of profiles proposed in each gossip. Yet, even when only 10 profiles are stored, at the end of the 200^{th} cycle, more than 68% of neighbors in the personal networks are identified. If the users provide sufficient storage, we observe that 50 cycles are enough to feed more than 90% of the personal networks.

3.2.2. Query Processing in Eager Mode. As described before, the queries are processed through the eager mode of P4Q. To evaluate the quality of the top-k results, we run a top-10 processing in a centralized implementation of our protocol and take the 10 returned items for each query as relevant items. The results obtained with P4Q are then compared to this baseline. The *recall* [Witten et al. 1999] R_k is then measured and computed as follows.

$$R_k = \frac{Number\ of\ Retrieved\ Relevant\ Items}{Total\ Number\ of\ Relevant\ Items}$$

Recall quantifies the coverage of the result set and varies between 0 and 1. In our experiments, we use average R_{10} over all queries as the results depend on the query and the user who generates it. In this context, an ideal $recall = 1$ means that all queries processed in P4Q achieve the same top-k results as the baseline.

Figure 7 depicts the evolution of the average R_{10} assuming each user stores 10 profiles in her personal network with different values of α. The smaller α, the larger portion of the remaining list is taken in charge by the gossip destination. If α is set to 0, the query is successively forwarded along a path away from the querier. This is similar to the traditional routing of queries in an unstructured P2P system [Bender et al. 2007]. In contrast, $\alpha = 1$ means only the neighbors of the querier are asked one by one. We vary the value of α to measure the efficiency of our protocol between the two extremes (Figure 7).

The average recall at cycle 0 corresponds to the top-10 results obtained by local processing with profiles in the personal networks. Encouragingly, with only 10 profiles, on average, more than 4 good items out of 10 can be returned without any gossip.

We observe from Figure 7 that the splitting factor α has an important impact on the top-k processing speed: $\alpha = 0.5$ outperforms other values and the closer α to 0.5, the faster the top-10 results approach the reference. This confirms our analytical measures.

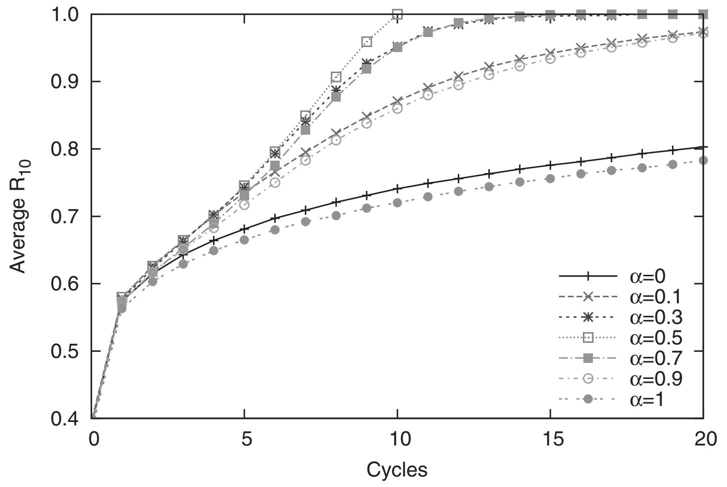


Fig. 7. Impact of splitting factor (α) on average recall ($c = 10$).

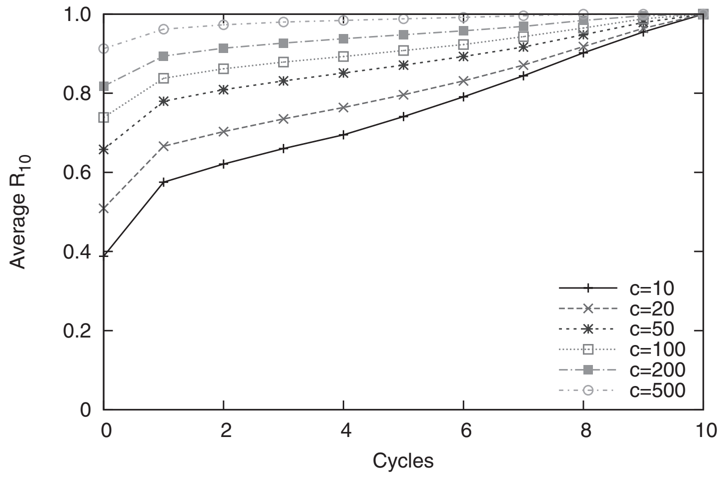


Fig. 8. Impact of stored profiles (c) on average recall ($\alpha = 0.5$).

Figure 8 depicts the latency of the top-*k* processing with $\alpha = 0.5$ assuming users store different profiles in their personal networks. At the end of the 10th cycle, all the queries get the most relevant results, that is, $R_{10} = 1$. Interestingly, the improvement in average recall after the first cycle is much more significant than that in the following cycles. This means that users with limited storage and little patience do not need to wait for a long time and the relatively satisfactory results can be obtained almost immediately.

As $\alpha = 0.5$ performs the best, 0.5 is considered the default value in the following evaluation. However, users still have the freedom to change that value if they have limited bandwidth or if they are willing to keep their personal networks more up-to-date. Detailed results will be presented later (Section 3.4.1).

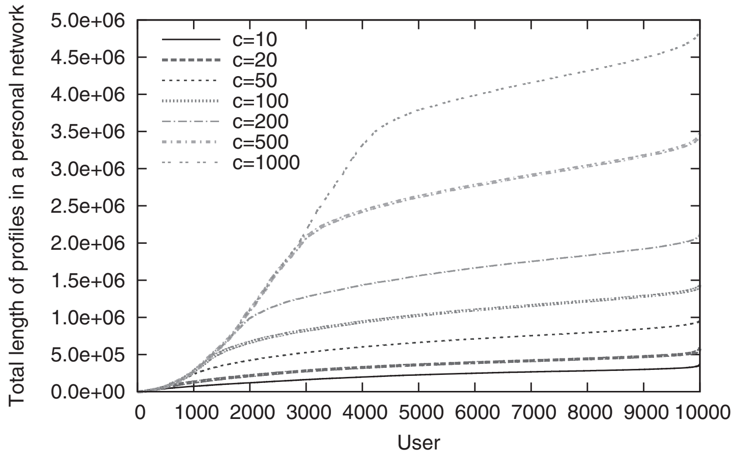


Fig. 9. Space requirement.

3.3. Cost Analysis

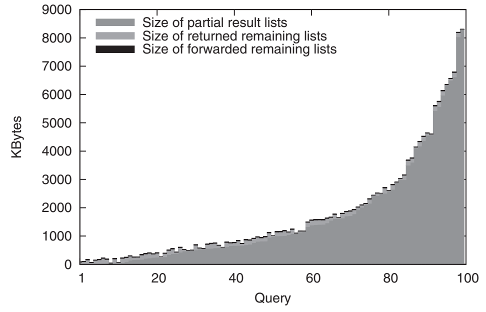
3.3.1. Storage Requirements. As opposed to the centralized personalized top-k processing approach presented in Amer-Yahia et al. [2008a] where the system stores all the user profiles and the related inverted lists, users in P4Q only store a limited number of their neighbors' profiles, significantly limiting the storage requirements.

Each user stores the profile digests of all its neighbors in her personal network and random view and the profiles of c closest neighbors. In our experiments, on average, each profile digests accounts for 0.78KB. The total storage required for the profile digests (1,000 in personal network and 10 in random view) is thus 787.8KB.

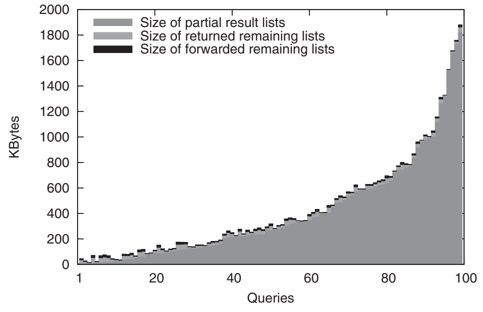
So the storage requirement is mostly determined by the size of the stored profiles, which in turn is strongly dependent on the contents of the profiles. We use a metric similar to that used in Amer-Yahia et al. [2008a] to measure the space requirement. The length of each profile is defined as the number of tagging actions it contains. The overall storage for the profiles in the personal network is then simply the sum of their lengths.

Figure 9 illustrates the storage requirement of each user for various numbers of stored profiles. Users are ranked in ascending order of their space requirements and the value on the x-axis can be simply considered as user identification. Obviously, the more profiles a user stores, the more space is required. Yet, if a user does not have a sufficient number of neighbors exhibiting similar interests with her, her storage remains the same even if she can store more. Note that storing 10 profiles requires only 6.8% of the space required to store all profiles in the personal network, while storing 500 requires 73.6% of that space. To illustrate this further, a single item (URL) in our trace is identified by its 16-byte MD5 hash value and each user has a 4-byte ID. We use 4-byte Unix time to describe the time in the system. Assuming that each tag can be identically presented as a 16-byte string, a tagging action takes 40B. Storing 10 profiles in the personal network requires only 13.8MB. This requirement can even be fulfilled by mobile devices with limited capacities.

3.3.2. Bandwidth Consumption. Due to the periodic behavior of lazy gossiping and the burst of communication generated by eager gossiping, data are continuously exchanged in the system. We now evaluate the bandwidth consumption of personal network maintenance and top-k processing. We concentrate on the two heterogeneous scenarios, namely the Poisson distribution with $\lambda = 1$ and $\lambda = 4$.



(a) λ = 1



(b) λ = 4

Fig. 10. Query processing bandwidth.

Personal Network Maintenance Traffic. As mentioned earlier, 50 profile digests are regularly transferred by each user having more than 50 profiles in her personal network. This imposes a transmission of 39KB for each user. Only 7.8KB and 15.6KB are transmitted for users having 10 or 20 profiles in their personal networks.

Except for the profile digests, the information transmitted by each user for maintaining her personal network consists of two parts: (i) the tagging actions to compute the exact similarity scores according to the estimation and (ii) the whole profiles to be stored in the personal network. The latter ones are only transmitted when better profiles are discovered. We trace the information exchanged by each user as the time passes in the scenarios with $\lambda = 1$ and $\lambda = 4$ respectively.

In the $\lambda = 1$ scenario, on average, at each cycle, before the personal networks stabilize, 74.6% of users in the system have to transmit further information for measuring the exact similarity while only 4.5% of them require the exchange of the whole profiles. For these users, 8.94KB and 586KB are transmitted respectively if they have such need in a given cycle. Practically, the maximum information transmitted in a single cycle does not exceed 5MB. Similar performance is observed in the $\lambda = 4$ scenario. On average, at each cycle, 79.6% users have to transmit 12.56KB for measuring the similarity while 15.3% of them need the whole profiles of 644KB. This is due to the fact that more neighbors could be identified at the same time while gossiping with a user having a large number of profiles in her personal network and more profiles are necessary to feed the personal network of a user having high storage capacity. In the bottom layer of the lazy gossip, 10 profile digests of 7.8KB are exchanged at each cycle. In fact, using the profile digests to estimate the similarity score upper bounds deters on average 13% users ($\lambda = 1$) and 8% users ($\lambda = 4$) from requiring additional information to compute the exact scores at each cycle, comparing to the case where any common item in the profile digest would require such computation (i.e., P3Q). This is due to the fact that unqualified users are immediately pruned after the upper-bound estimation. This brings a save of 6.91KB ($\lambda = 1$) and 11.12KB ($\lambda = 4$) per cycle for each user.

Query Processing Traffic. When a query is gossiped in the system, three kinds of information are transmitted: the forwarded remaining list, the returned remaining list, and the partial result lists returned to the querier along with users whose profiles are used to build these lists.

In our experiments, a user is identified by a 4-byte ID and the version of her profile is identified by a 4-byte integer. The score of each item in the partial result list can also be presented by a 4-byte integer. Figure 10 depicts the quantity of information transmitted to answer a query in the scenarios with $\lambda = 1$ and $\lambda = 4$ respectively. For the sake of visibility of the figure, only 100 queries, randomly picked from the whole set of queries, are shown. The values on the y-axis represent the sum of the information transmitted by all the users reached by the query during the query processing period. Users are ranked in ascending order of the quantity of partial result lists which consume most of the bandwidth compared to other information. The value on the x-axis represents an individual query.

On average, in the $\lambda = 1$ scenario, 573KB are transmitted to answer a query and in the $\lambda = 4$ scenario, 360KB are transmitted. The reason is that in a system where many users have large storage capacity, several profiles involved in a user's query could be found through a single user. This prevents different users from transmitting the same items appearing in different profiles. It is worth noticing that using the Bloom filter of tags in the profile digest, on average, the initial forwarded remaining list accounts for only 87% of its length if no such information is available in both $\lambda = 1$ and $\lambda = 4$ scenarios (i.e., P3Q). This further confirms the benefits of using the profile digest *ItemTagBloom* in P4Q.

Note that the remaining lists are piggybacked in the eager gossip messages and do not generate additional messages in the system. In contrast, each partial result list is sent to the querier in a separate message. On average, to answer a query, 230 such messages are transferred to the querier in the $\lambda = 1$ scenario and 71 in the $\lambda = 4$ scenario. As a result, the size of each message is in fact very small. This also verifies the bound on the number of partial result lists of the analysis (Theorem 2.3).

3.4. Dynamics

Users in collaborative tagging systems are usually active in the sense that they change their profiles frequently by tagging new items. In addition, new users keep joining the system and users are not online at all times. We evaluate in this section the impact of both forms of dynamics, respectively the profile dynamics and the users churn (users leaving) on the same delicious trace.

3.4.1. Profile Dynamics. First, we analyze the underlying patterns of changes in the system during the whole year of 2008. We observe that every week, more than 3,000 users change their profiles while less than 60 new users are involved in the system on average. As the modification of profiles dominates the arrival of new users, we focus on the impact of changes on user profiles. Note that the number of cycles for new users to build their own personal networks should be similar but smaller than the case where no user exists in the system before, as shown in Figure 6(b). Our analysis also shows that the number of users changing their profiles per week remains stable. We take the week having the largest variation (from 2008-11-11 to 2008-11-18) to run the simulation. We assume that all users change their profiles simultaneously, that is, each user adds the new tagging actions on the same day to her profile at the same moment of the simulation. The simulations are run for each day in this week, but only one of them is shown as they all exhibit similar trends.

Updating Profiles. A user profile may be replicated in different personal networks. When a profile is updated, the changes are captured through the gossip protocol. To evaluate the ability of P4Q to capture such changes, we consider the Average Update Rate (AUR) as a measure of the freshness of the profiles in the users' personal networks at a given cycle. The update rate for a user is defined as the number of profiles in her personal network that have been updated over the number of profiles that have been subject to changes. The average update rate is averaged over all users, that is,

$$AUR = \frac{1}{|U|} \sum_{u_x \in U} \frac{\textit{Number of Updated Profiles in Network}(u_x)}{\textit{Number of Profiles in Network}(u_x)\ \textit{Owing Update}}.$$

AUR reaches 100% when the profiles in all users' personal networks are up-to-date.

Table II. Impact of Profile Changes in Different Systems

Number of stored profiles (c)	Fraction of users having to update profiles	Average number of profiles to update	Maximum number of profiles to update
10	80.9%	4	10
20	82.0%	7	16
50	88.2%	15	34
100	88.2%	26	61
200	88.2%	43	106
500	88.2%	76	224
1,000	88.2%	105	388

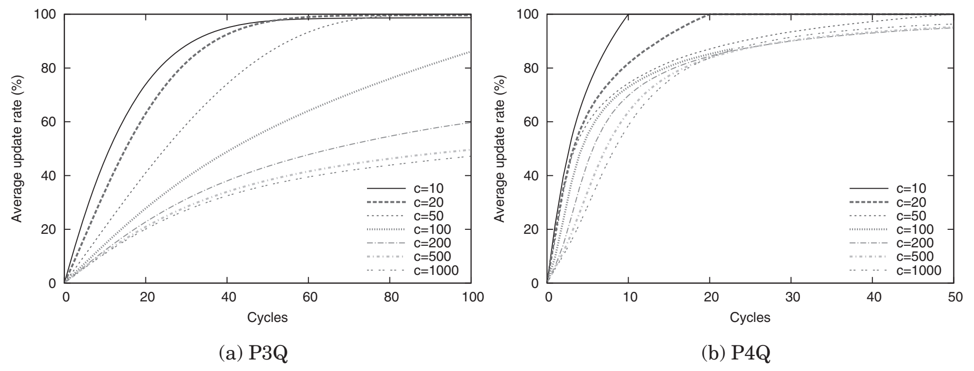


Fig. 11. AUR evolution in lazy mode with uniform stored profiles (c).

To highlight the impact of storage on the evolution of the average update rate, the simulations are first run in homogeneous settings where all users have the same number of profiles (c) in their personal networks. We consider the day where 1,540 users changed their profiles with an average of 8 new tagging actions per profile. Maximum change was observed in a profile with 268 new tagging actions. Table II summarizes the influence of profile changes in different settings. Note that we focus on the average update rate of the replicated (and outdated) profiles but ignore their impact on the query result quality. In fact, the profile change of the 1,540 users only influences the top-10 results of 0.5% queries, that is, if no profile is updated according to this change, 0.5% queries miss at least one item introduced by the new tagging actions in their top-10 results. Despite its small impact, accumulating such profile changes may significantly hurt the result quality in the long term. We are thus interested in how the changes can be gradually captured through both the lazy and the eager modes of P4Q.

In lazy mode, that is, after users changing their profiles, no query is generated; we compute the average update rate after each cycle. To emphasize the ability of P4Q to enable efficient profile updating by using self-promotion and mutual-aid, we also compare the AUR with the case where users always gossip with each other in the same way even if the changes occur (i.e., P3Q). We observe from Figure 11(a) that a small number of stored profiles (c) guarantees a high average update rate. After 30 cycles, more than 95% profiles are updated in both systems for users storing 10 or 20 profiles while only 40% of the profiles are updated after 100 cycles for the users storing 500 or 1,000 profiles. Not surprisingly, the more profiles are stored in a personal network, the more difficult it is to keep all of them up-to-date.

However, if the users actively react to the changes as described in P4Q, we observe from Figure 11(b) that the profile updating is significantly accelerated. If each user stores only 10 profiles, all the users can update their stored profiles within 10 cycles.

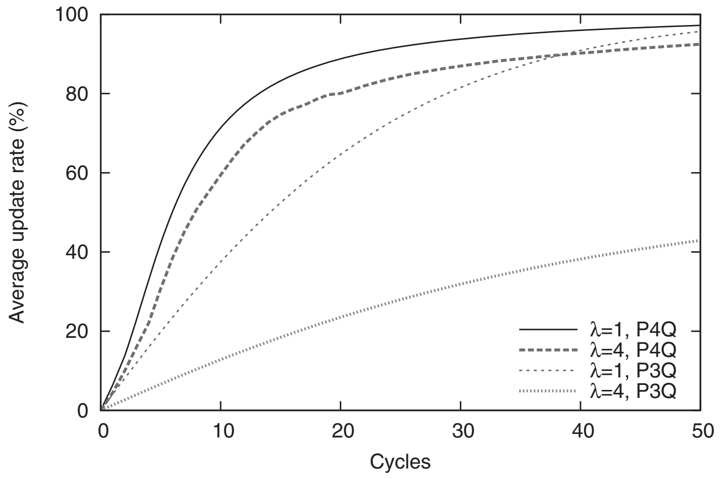


Fig. 12. Impact of stored profiles (c) distribution on AUR in lazy mode.

Encouragingly, even if each user stores 500 or 1,000 profiles, more than 80% profiles are updated within 20 cycles. This significant improvement in updating rate comes from the fact that users are more willing to promote their own changes (self-promotion) and share with other users their up-to-date information (mutual-aid).

Similarly, in the heterogeneous scenarios with $\lambda = 1$ and $\lambda = 4$, Figure 12 confirms the former observation that if most of the users in the system store a small number of profiles, it is easier to keep them up-to-date. In fact, gossiping in the P4Q allows appealing updating rate regardless of the distribution of profiles stored in each user's personal network.

We now consider the impact of running the eager mode on the freshness of the system. The lazy mode guarantees that the personal networks are updated uniformly across users as the gossip protocol runs periodically on every user. Instead, the eager mode runs on-demand, upon query, and impacts a small portion of the network, that is, the small fraction of users reached by the query and contributing to the query processing. This has a significant impact on the freshness of the personal networks of such users. To illustrate the ability of the eager mode to cope well with dynamics, we compute the average update rate over the users participating in the eager gossip. The number of such users reached by the query in the two heterogeneous scenarios is shown in Figure 13. The x-axis captures the query identification and the queries are ranked in descending order of their y-axis values. On average, during the processing of a single query, 219 users are reached by the query in the $\lambda = 1$ scenario while 66 users are reached in the $\lambda = 4$ scenario. With the help of the tag information contained in the profile digests, the number of neighbors to collect during the eager mode is reduced. As a result, instead of gossiping a list of $s - c$ neighbors (i.e., P3Q), only a subset of users who have used at least one tag in the query are gossiped. This leads to an average reduction of 37 users ($\lambda = 1$) and 9 users ($\lambda = 4$) reached by the query.

Figure 14 shows the impact of the eager gossip on profile updating. To see a significant impact, a series of queries are consecutively sent by the same user before the next cycle of lazy gossip begins. Here we also compare P4Q against its initial version P3Q where all users gossip in the same way regardless of the profile changes. We observe that if most users have small storage capacity ($\lambda = 1$), the acceleration effect of eager gossip is prominent. After answering a single query, on average, about 50% profiles are updated in P4Q. Ten consecutive queries enable all the users reached by the queries to update

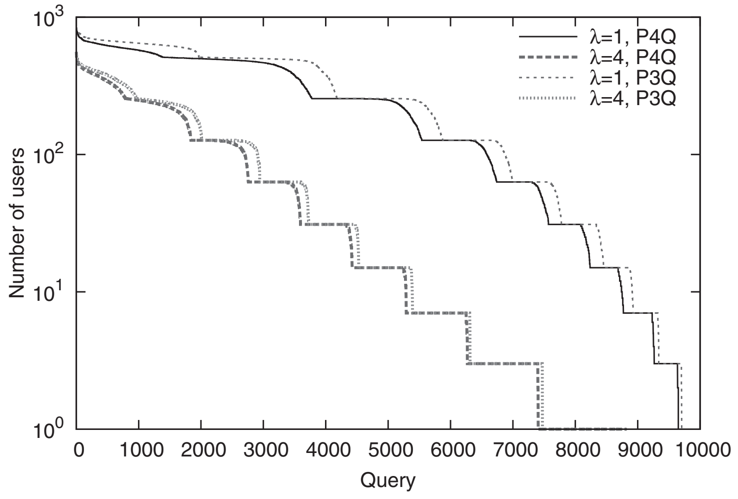


Fig. 13. Number of users reached by the query.

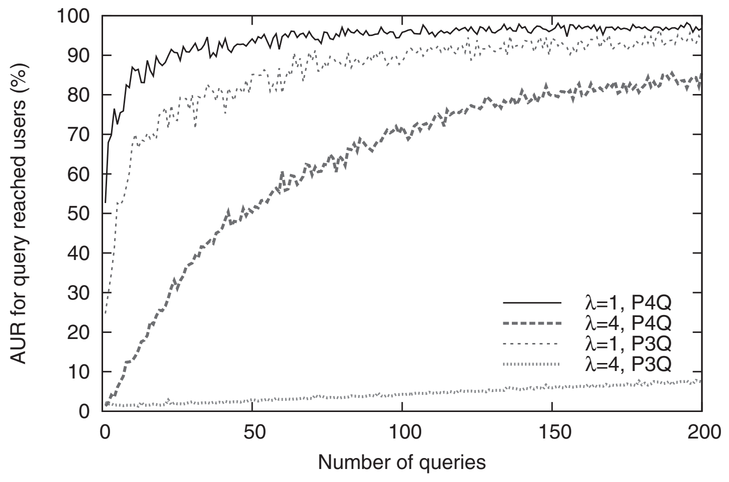


Fig. 14. AUR evolution in eager mode.

more than 85% of the changed profiles. Yet, all the changes are not taken into account only relying on the eager mode. This is due to the fact that in the absence of the lazy gossip, changes of users that are not reached by the queries are not yet propagated. This also explains why the impact of eager gossip is less significant when users have large storage capacity. Moreover, with small storage capacity, in each cycle of gossip, the same profiles are proposed. Once a profile is updated, the gossip protocol ensures a fast dissemination.

It is worth noticing that in P4Q, when the users have large storage capacity ($\lambda = 4$), gossiping in eager mode with mutual-aid significantly outperforms its initial version P3Q. Answering 10 consecutive queries allows the users participating in the query processing to update more than 10% of their stored profiles, which, as we observe from Figure 14, is very difficult if users do not actively react to the changes.

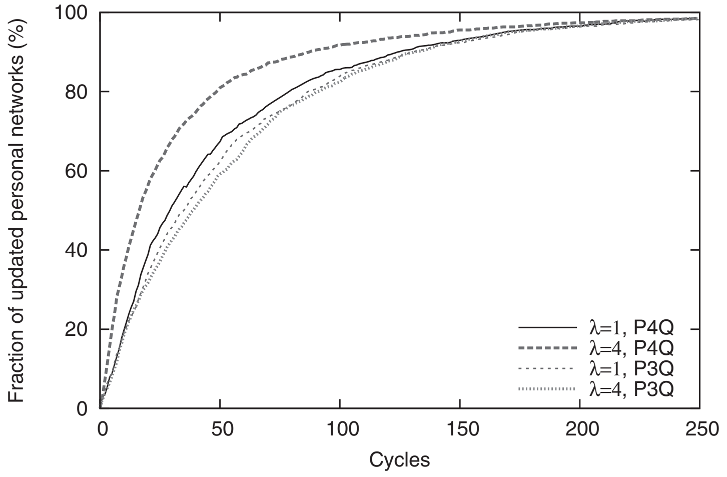


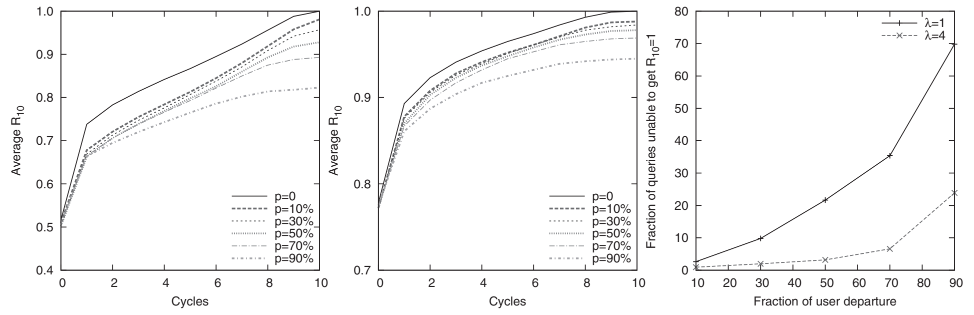
Fig. 15. Personal network evolution in lazy mode.

Updating Neighbors. Active tagging behaviors of users may not only impact the stored profiles, but also the personal networks themselves. Considering the same day in the simulation, we observe that the changes in profiles led to 1, 719 users changing an average of 2 (maximum 148) neighbors in their personal networks. We now evaluate how fast such changes are captured under the lazy mode. To this end, we compute the ratio of users discovering all their new neighbors over the users whose personal networks should change. Note that this is a strict metric in the sense that even when most of a user's new neighbors are discovered, the ratio is still 0 unless her personal network is completed.

Figure 15 shows that, if users do not actively react to profile changes (as in P3Q), in both settings, after 30 cycles, half of the users have discovered all their relevant neighbors and at the 100^{th} cycle, the number reaches 80%. Yet, P4Q provides even better performance: if most users have a large storage capacity ($\lambda = 4$), less than 20 cycles are needed to provide 50% users with their new personal networks while 80% users find all their new neighbors using half the cycles required in P3Q. Note that in P4Q, users find new neighbors faster if they have large storage capacity. This is because only a handful of new tagging actions are added to the profiles each day, and new neighbors are more likely to have low similarity scores when compared to the existing neighbors. When users have small storage capacity, they only keep the profiles having the highest similarity scores in their personal networks. Exchanging such profiles between two users makes it difficult to find the neighbors that are less similar. However, benefiting from the more up-to-date profiles when users have small storage capacity, the new trends in their personal networks can be efficiently captured, which is also slightly more efficient than in P3Q.

We do not display the results for the eager mode. Effectively the eager mode does not impact the neighbor discovery as the gossip operations are limited to the querier's neighborhood.

3.4.2. Churn. Users who do not store all their neighbors' profiles should collect more information through gossiping. However, the original owner of a profile may not be on-line at query time. We now evaluate the failure-resilience capability of P4Q. Inherently, the fact that users store several profiles in addition to their own profiles guarantees a minimum number of replicas of each profile in the system. Moreover, if the owner of



(a) similarity score distribution (b) personal network convergence (c) query processing ($\lambda = 1$)

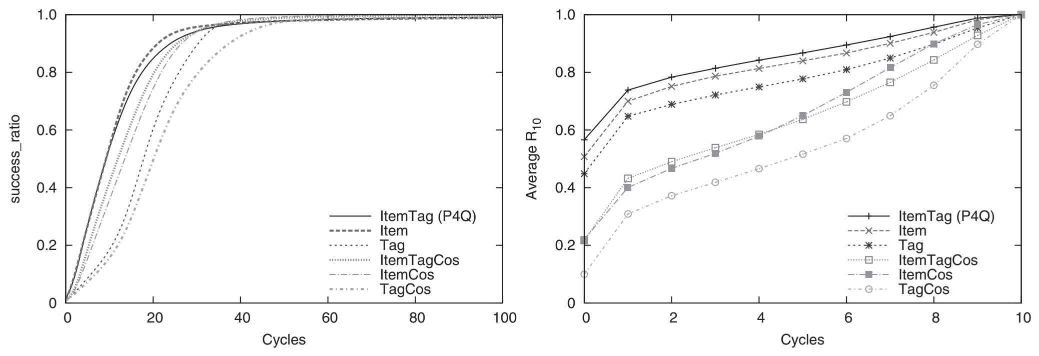
Fig. 16. Impact of user departure on top-k.

a given profile has left the system, the replicas of her profile would not be out-of-date. Effectively, her opinion on the tagged items remains meaningful and no new tagging actions can be added to her profile during her absence. However, the departure of a large number of users will inevitably cause problems. More specifically, this will influence the query processing time as more users should be contacted to get the necessary profiles but also the top-k quality, as some profiles might no longer exist in the system.

Unfortunately, no information regarding the online time of each user could be obtained by crawling a delicious trace. So we simply assume that a given percentage of randomly chosen users leave the system simultaneously. Figure 16 illustrates the impact of the number of leaving users on the top-k processing in the $\lambda = 1$ and $\lambda = 4$ scenarios respectively. We denote the percentage of leaving users by p. Obviously, the more users leave the system, the slower the average recall improves along time. However, even if 90% users have left, at the end of the 10^{th} cycle, on average, about 8 relevant items can be returned to the querier in the $\lambda = 1$ scenario (Figure 16(a)). Better results are observed in the $\lambda = 4$ scenario (Figure 16(b)). This is due to the fact that in the latter system, more replicas are available thanks to larger storage capacity of the remaining users. If only 10% of the users leave, the degradation on processing time is negligible. Yet, the average recall fails to get 1 regardless of how long the users wait because a certain number of queriers cannot find all the profiles in their personal networks (Figure 16(c)). However, even if 50% of the users leave simultaneously in the $\lambda = 4$ scenario, the percentage of noncomplete queries remains smaller than 5%. Those results confirm that our system is robust in the face of user departures: after waiting for a limited time (10 cycles), almost all the relevant items can be proposed to the querier.

3.5. Generalization and Scalability

3.5.1. Generalization. In P4Q, we consider the number of common tagging actions ($\langle i, t\rangle$) as the measure of similarity between users. Here we evaluate P4Q in terms of personal network maintenance in lazy mode and query processing in eager mode to convey the applicability of alternative similarity metrics. To this end, we consider 5 different metrics, well-known in the context of collaborative tagging systems [Amer-Yahia et al. 2008b; Noll and Meinel 2007], to measure the similarity between users. These metrics explore the similarity on items, tags, or $\langle item, tag\rangle$ pairs involved in users' tagging behaviors and fall into two categories, *nonnormalized* and *normalized*, with respect to users' overall tagging behaviors.



(a) personal network convergence in lazy mode (b) query processing in eager mode ($\lambda = 1$)

Fig. 17. Performance of P4Q with different similarity metrics.

The first similarity score, referred to as *Item*, counts the number of common items between two users, that is,

$$Item_{u_x}(u_y) = |\{i|\ Tagged_{u_x}(i, *, *) \wedge Tagged_{u_y}(i, *, *)\}|.$$

The second similarity score, referred to as *Tag*, uses the number of common tags between two users, that is,

$$Tag_{u_x}(u_y) = |\{t|\ Tagged_{u_x}(*, t, *) \wedge Tagged_{u_y}(*, t, *)\}|.$$

The other three measures are normalized using the cosine similarity on items, tags, or $\langle item, tag\rangle$ pairs respectively. More specifically, the similarity score *ItemCos* is computed as

$$ItemCos_{u_x}(u_y) = \frac{|\{i|\ Tagged_{u_x}(i, *, *) \wedge Tagged_{u_y}(i, *, *)\}|}{\sqrt{|\{i|\ Tagged_{u_x}(i, *, *)\}|} \cdot \sqrt{|\{i|\ Tagged_{u_y}(i, *, *)\}|}}.$$

The similarity score *TagCos* is computed as

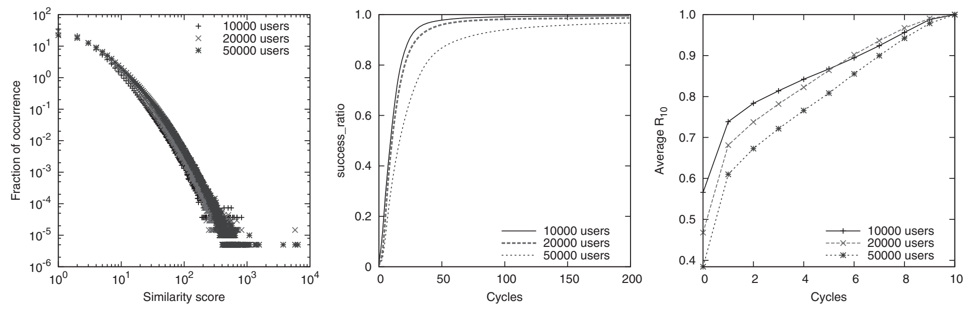
$$TagCos_{u_x}(u_y) = \frac{|\{t|\ Tagged_{u_x}(*, t, *) \wedge Tagged_{u_y}(*, t, *)\}|}{\sqrt{|\{t|\ Tagged_{u_x}(*, t, *)\}|} \cdot \sqrt{|\{t|\ Tagged_{u_y}(*, t, *)\}|}}.$$

Finally, the similarity score *ItemTagCos* is computed as

$$ItemTagCos_{u_x}(u_y) = \frac{|\{\langle i, t\rangle|\ Tagged_{u_x}(i, t, *) \wedge Tagged_{u_y}(i, t, *)\}|}{\sqrt{|\{\langle i, t\rangle|\ Tagged_{u_x}(i, t, *)\}|} \cdot \sqrt{|\{\langle i, t\rangle|\ Tagged_{u_y}(i, t, *)\}|}}.$$

As in P4Q, for all these scores, increasing value indicates higher similarity and each user keeps the *s* users having the highest scores to form her personal network.

Figure 17(a) compares the convergence property of the personal networks in lazy mode of P4Q with these similarity scores to that with the originally used similarity score, referred to as *ItemTag*. We assume that each user has uniform storage for her personal network of 1,000 users using the same dataset as described in Section 3.1 and gossips 50 profiles at each cycle. We observe from this figure that generally the nonnormalized similarity metrics ensure higher convergence speed than the corresponding normalized similarity metrics. This is due to the fact the clustering coefficient in the network formed with nonnormalized similarity metrics is higher. Interestingly, regardless of the specific similarity metric used in P4Q, the convergence speed of the personal



(a) similarity score distribution (b) personal network convergence (c) query processing ($\lambda = 1$)

Fig. 18. Performance of P4Q with different system scale.

networks is quite similar: at most 50 cycles are enough to feed more than 99% of the personal networks.

Figure 17(b) further compares the efficiency of the query processing in eager mode of P4Q, when different similarity metrics are used to form the personal networks. We assume that the storage of users follows the Poisson distribution with $\lambda = 1$ (Table I). Not surprisingly, the top-10 results of the queries improve faster when the personal networks are formed with nonnormalized similarity metrics. This is because higher clustering coefficient implies that the desired profiles for the query processing are more likely to be discovered in the querier's close neighborhood. Yet, as proven by our analysis in Section 2.4, at the end of the 10^{th} cycle, the accurate top-10 results are obtained with all these similarity metrics.

In fact, as we have seen from this experiment, P4Q is generic in the sense that alternative similarity metrics can be easily applied to form personal networks and its main performances hold anyway.

3.5.2. Scalability. The evaluations reported earlier are conducted in a system of 10,000 users, where each user maintains a personal network of 1,000 neighbors. Using such a large personal network guarantees the diversity of the items involved in each user's personal network, which in turn guarantees the personalized query result quality. In fact, when the system grows, users do not need to grow their personal networks to obtain the query results of comparable quality. Figure 18(a) depicts the distribution of the similarity scores between each user and her neighbor in the systems of 10,000, 20,000, and 50,000 users respectively, where each user has a personal network of 1,000 neighbors.[6] We observe that given the same size of personal network, users tend to select users with higher similarity scores as their neighbors in a system of larger scale. Since users having similar behaviors to the querier are more likely to provide satisfactory results [Amer-Yahia et al. 2008b], this increase of similarity scores implies that the appropriate size of personal network does not increase with the scale of the system. We then focus on the performance of P4Q in terms of the personal network maintenance in lazy mode and the query processing in eager mode, with each user having 1,000 neighbors in her personal network.

Figure 18(b) depicts the convergence speed of the personal networks in the three systems, where each user locally stores all the profiles in her personal network and gossips 50 profiles at each cycle. Generally, increasing the scale of the system requires longer time to build the personal networks. Yet, we observe that doubling the size of

[6]The dataset used in this experiment is crawled in the same way as introduced in Section 3.1 but contains 137,898 users. The users in the systems with different scales are randomly selected from this dataset.

Table III. Statistics of Network Latency between a User-Neighbor Pair

Average	Max	Min	Standard deviation
70 ms	122 ms	2 ms	19 ms

the system has almost no impact on the convergence property of P4Q. Even in a five times larger system (50,000 users), after 50 cycles, the personal networks are still filled by about 90% of the appropriate neighbors. We ignore the performance for updating the personal network when users' interests change as we have seen previously in Figure 6(b) and Figure 11(b), updating the personal networks is more efficient than building them from the very beginning where each user joins the system and only knows some random users as neighbors.

Figure 18(c) further compares the efficiency of the top-10 processing in these systems, assuming the storage of users follows the Poisson distribution with $\lambda = 1$ (Table I). We observe from this figure that the average recall improves faster in a smaller system. Since users are more clustered in a smaller system given the same number of neighbors in their personal networks, it is easier for the querier to discover the profiles not locally stored through gossiping. Encouragingly, compared to a system of 10,000 users, the average recall degrades by at most 20% at the very beginning in a system of 50,000 users and reaches 1 at the end of the 10^{th} cycle as well.

These results convey the scalability of P4Q: increasing the scale of the system does not impose much overhead to maintain the personal networks and a limited number of cycles are enough to satisfy the queries regardless of the system scale. In fact, as shown by our analysis (Section 2.4), the scalability of P4Q stems from the fact that the query processing time is bounded by the size of the personal network and is independent on the size of the entire system.

3.6. P4Q in Practice

Our evaluation demonstrates that the users get fairly good results immediately. Those results can be refined collaboratively until accurate results are provided within a small number of gossip cycles. Although P4Q takes more time to build the personal networks, if the users store less profiles, once most of the neighbors are identified, P4Q guarantees a better freshness of the local stored information and consumes less bandwidth for the personal network maintenance. Yet, users still have the possibility to store more information if they are willing to get better results immediately.

To convey the feasibility of P4Q in a real system, we use the GT-ITM transit-stub model [Zegura et al. 1996] to generate an underlying network topology in order to quantify the network latency between users gossiping with each other. The generated topology consists of 10,304 nodes, where 16 of them are transit nodes. Link delays between two transit nodes, a transit node and a stub node, and two stub nodes are chosen uniformly between [15, 25] ms, [5, 9] ms, and [2, 4] ms, respectively [Tan and Jarvis 2007]. 10,000 stub nodes are randomly selected and mapped to the 10,000 users in P4Q. Table III summarizes the statistics of the network latency between each user and each neighbor in her personal network, obtained by solving the all-pair shortest path problem on the generated topology. As we can see, the maximum latency between a user and her neighbor does not exceed 122 ms. Real latency measures performed on the Internet, such as the ones presented in Dabek et al. [2004], confirm this observation: the median round-trip latency on PlanetLab is 76 ms while 99% of them are under 400 ms.

Assuming 1 minute per cycle and 5 seconds per cycle are used in the lazy mode and the eager mode of P4Q respectively, the network latency of tens of milliseconds has almost no impact on the performances of P4Q: the communication between a user and her neighbor can be efficiently established and even rebuilt in case of broken links.

Considering, for instance, the scenario with $\lambda = 1$, the query can be accurately answered within 50 seconds with an average bandwidth consumption of 91Kbps (Kbits per second) for the querier.[7] The background traffic for maintaining the personal network through lazy gossip is only 7.6Kbps and this may increase to 79.2Kbps in eager gossip. Knowing that nowadays ADSL supports several Mbps of bandwidth, these only account for less than 1% of that amount.

In fact, if the system that applies P4Q to provide personalized top-*k* processing tolerates more bandwidth consumption, both the lazy mode and the eager mode can run at higher frequency, which would significantly decrease the personal network construction time as well as the query processing time. For instance, in Frey et al. [2009], a gossip-based protocol is deployed on PlanetLab to support video streaming and 600Kbps stream are successfully disseminated with a gossip cycle of 200 ms. If the same bandwidth is provided in P4Q, the eager gossip can run every 0.6 seconds, significantly reducing the query processing time. Finally, even if all users simultaneously change their profiles, in half an hour, 95% of the local stored information is updated and more than 50% of users' new neighbors are identified.

4. CONCLUDING REMARKS

The importance for future search engines to leverage (either explicit or implicit) context information to improve the search process was pointed out in 2000 [Lawrence 2000]. This was also confirmed in Teevan et al. [2007] where personalization was considered a promising way to boost the quality of search engines.

Two general approaches for search personalization were described in Pitkow et al. [2002]: query expansion and result processing. The first approach appends new terms to a query in order to better reflect the user's profile [Carman et al. 2008; Chirita et al. 2007]. The second runs the original query but reranks the returned results based on the user's profile. A wide range of user activities have also been considered to enhance reranking, including query histories [Speretta and Gauch 2005; Dou et al. 2007], browsing histories [Sugiyama et al. 2004], and tagging behaviors [Noll and Meinel 2007].

Various community-aware ranking algorithms have been developed to explore the relationships between users in personalizing information retrieval. A social scoring function, leveraging the strength of user relations and correlations among different tags, was proposed in Schenkel et al. [2008] to improve the top-*k* quality. Various notions of user affinities and social relations were also discussed in Amer-Yahia et al. [2008b] and Schenkel et al. [2008]. A general indexing and query processing framework, encompassing a wide class of scoring functions and networks, was developed in Amer-Yahia et al. [2008a]. Given a user and a so-called user's network, the relevance of an item to the user's query is a function of its popularity in that network. It is, however, shown that building inverted lists for each $\langle user, tag \rangle$ pair is space-intensive, while clustering users with similar tagging behaviors and building inverted lists for each cluster impacts the processing time.

The way P4Q performs the top-*k* processing is inspired by the network-aware search technique of Amer-Yahia et al. [2008a]. P4Q is, however, decentralized and gossip-based, and this we believe is the key to its scalability, in terms of both storage and processing. Several approaches to decentralize top-*k* processing have been proposed. In Michel et al. [2005], precomputed inverted lists are distributed across nodes and

[7] 50 seconds can be considered as a long response time. Yet, this is the response time in the worst case, which ensures all queries obtain recall 1 regardless of the scale of the system. In fact, most of the queries can be accurately computed at the first few cycles. Letting the eager gossip run every 5 seconds also gives queriers the possibility to check the intermediate results and terminate the processing earlier.

partial information, approximating top-*k* results, is progressively transmitted in the network. In Bender et al. [2005], a Chord-based DHT is used to partition the term space and each node is responsible of a random set of terms. The query is then routed to the nodes responsible for the query-related terms. These approaches differ from P4Q as they rely on a global dictionary or specific mechanisms to organize the data.

SPEERTO [Vlachou et al. 2008] explores a skyline-based routing which forwards the top-*k* queries among supernodes to minimize the data transferred in the system. PlanetP [Cuenca-acuna et al. 2003] uses gossip to globally replicate a membership directory and a term-to-peer context index. A searching node first identifies the set of nodes with terms related to the query and then ranks the relevant documents (returned by these nodes) to determine the most pertinent ones. Unlike P4Q, none of these approaches achieves personalization.

In the context of top-*k* processing, explicit (declared) social connections have also been considered. The potential for using social networks to enhance Internet search was discussed in Mislove et al. [2006]. The proposed system, PeerSpective, focuses on a small set of social friends (in Skype or in Lab) to rank relevant results. In fact, equipping each P4Q user with a predefined explicit network (e.g., Facebook) as input would be straightforward: only the eager mode of P4Q would suffice.

P4Q relies on a gossip-based protocol to discover and leverage implicit relations in order to provide a personalized query processing scheme for large-scale systems. Thanks to its simplicity, flexibility, and robustness, the gossip-based communication paradigm has been applied in many settings such as information dissemination [Eugster et al. 2004], aggregation [Jelasity 2004], and overlay topology management [Voulgaris and van Steen 2005].

eXO [Loupasakis et al. 2011] supports social networking services as well as personalized top-*k* processing in peer-to-peer systems. While P4Q is maintained through gossip-based protocols, eXO relies on a DHT to organize the tagging profiles and process the queries. Each tagging action is indexed in the DHT with a copy of the user profile. Although eXO is able to answer any query, be they related or not to the querier's profile, its rigid structure sacrifices load balancing by imposing a high load on nodes responsible for popular tags. On the contrary, P4Q is more flexible and balanced, and focuses on personalized queries related to user profiles.

Finally, it is important to note that P4Q can be viewed as a refined and extended version of a preliminary protocol, presented in the conference version of this work [Bai et al. 2010]. In comparison with the preliminary protocol, we have extended this article as follows.

—We introduce a new similarity score upper bound estimation mechanism during the gossip of profiles based on different types of profile digests. This significantly reduces the bandwidth consumption for computing exact similarity scores between users.
—Instead of using a uniform profile digest for each user, we now adaptively adjust the size of each profile digest according to the size of the profile. While guaranteeing a low level of false positive rate for the estimation, we reduce both the space for storing the profile digests and the bandwidth for transmitting them.
—We encode the Bloom filter of tags in the profile digest and this improves the query processing in the eager gossip mode by pruning earlier the unqualified neighbors from the remaining list. This reduces the number of messages generated in the system for processing each query.
—We introduce a response mechanism that actively copes with the profile changes of each user and guarantees efficient personal network refreshment in terms of updating the stored profiles and discovering new neighbors.

—We introduce the notion of "time" into the tagging actions (profiles) as well as the queries. This new timestamp-based mechanism guarantees the consistency between the desired and obtained top-*k* results in face of profile updating.
—We generalize the application of P4Q to a wide range of similarity metrics that establish the implicit relations among users. This reveals its general applicability in peer-to-peer systems to support personalized query processing.
—We convey the scalability of P4Q by conducting experiments in larger systems and showing its efficiency for both maintaining the personal networks and processing the queries.
—We evaluate the network latency that P4Q would face and discuss its feasibility in a real system.

ACKNOWLEDGMENTS

The authors would like to thank Marin Bertier for gathering the data and the many helpful discussions they had, and Guang Tan for sharing his experience on characterizing the system features. The authors would also like to thank all the anonymous reviewers for their valuable comments which helped improving the quality of the article.

REFERENCES

Amer-Yahia, S., Benedikt, M., Lakshmanan, V., and Stoyanovic, J. 2008a. Efficient network aware search in collaborative tagging sites. In *Proceedings of the 34st International Conference on Very Large Databases (VLDB'08)*. 710–721.

Amer-Yahia, S., Marlow, C., Yu, C., and Stoyanovich, J. 2008b. Leveraging tagging to model user interests in del.icio.us. In *Proceedings of the AAAI International Conference on Social Information Proceeding (AAAI SIP'08)*.

Bai, X., Bertier, M., Guerraoui, R., and Kermarrec, A. 2009. Toward peer-to-peer personalized top-k processing. In *Proceedings of the 2nd ACM EuroSys Workshop on Social Networks Systems (SNS'09)*. 1–6.

Bai, X., Bertier, M., Guerraoui, R., Kermarrec, A.-M., and Leroy, V. 2010. Gossiping personalized queries. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT'10)*. 87–98.

Bender, M., Crecelius, T., Kacimi, M., Miche, S., Xavier Parreira, J., and Weikum, G. 2007. Peer-to-peer information search: Semantic, social, or spiritual? *IEEE Data Engin. Bull. 30,* 2, 51–60.

Bender, M., Michel, S., Triantafillou, P., Weikum, G., and Zimmer, C. 2005. MINERVA: Collaborative P2P search. In *Proceedings of the 31st International Conference on Very Large Databases (VLDB'05)*. 1263–1266.

Bloom, B. H. 1970. Space/Time trade-offs in hash coding with allowable errors. *Comm. ACM 13,* 7, 422–426.

Carman, M., M.Baillie, and Crestani, F. 2008. Tag data and personalized information retrieval. In *Proceedings of the CIKM Workshop on Search in Social Media (SSM'08)*. 27–34.

Chirita, P. A., Firan, C. S., and Nejdl, W. 2007. Personalized query expansion for the web. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'07)*. 7–14.

Cuenca-acuna, F. M., Peery, C., Martin, R. P., and Nguyen, T. D. 2003. PlanetP: Using gossiping to build content addressable peer-to-peer information sharing communities. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*. 236–246.

Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M. F., and Morris, R. 2004. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*. 85–98.

Dou, Z., Song, R., and Wen, J.-R. 2007. A large-scale evaluation and analysis of personalized search strategies. In *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*. 581–590.

Eugster, P. T., Guerraoui, R., Kermarrec, A. M., and Massoulie, L. 2004. Epidemic information dissemination in distributed systems. *IEEE Comput. 37,* 5, 60–67.

Fagin, R. 2002. Combining fuzzy information: An overview. *SIGMOD Rec. 31,* 2, 109–118.

Frey, D., Guerraoui, R., Kermarrec, A., Monod, M., and Quéma, V. 2009. Stretching gossip with live streaming. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'09)*. 259–264.

Jelasity, M. 2004. An approach to massively distributed aggregate computing on peer-to-peer networks. In *Proceedings of the 12th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP'O4)*. 200–207.

Jelasity, M., Montresor, A., Jesi, G., and Voulgaris, S. 2004. The Peersim simulator. http://peersim.sf.net.

Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A., and van Steen, M. 2007. Gossip-Based peer sampling. *ACM Trans. Comput. Syst. 25,* 3, 8.

Lawrence, S. 2000. Context in web search. *IEEE Data Engin. Bull. 23*, 25–32.

Loupasakis, A., Ntarmos, N., and Triantafillou, P. 2011. eXO: Decentralized autonomous scalable social networking. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR'11)*. 85–95.

Michel, S., Triantafillou, P., and Weikum, G. 2005. KLEE: A framework for distributed top-k query algorithms. In *Proceedings of the 31st International Conference on Very Large Databases (VLDB'05)*. 637–648.

Mislove, A., Gummadi, K., and Druschel, P. 2006. Exploiting social networks for internet search. In *Proceedings of the 5th ACM Workshop on Hot Topics in Networks (HotNets'06)*. 79–85.

Mislove, A., Marcon, M., Gummadi, K., Druschel, P., and Bhattacharjee, B. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC'07)*. 29–42.

Montresor, A. and Jelasity, M. 2009. PeerSim: A scalable P2P simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer (P2P'09)*. 99–100.

Noll, M. and Meinel, C. 2007. Web search personalization via social bookmarking and tagging. In *Proceedings of the 6th International and 2nd Asian Semantic Web Conference (ISWC'07 + ASWC'07)*. 365–378.

Pitkow, J., Schütze, H., Cass, T., Cooley, R., Turnbull, D., Edmonds, A., Adar, E., and Breuel, T. 2002. Personalized search. *Comm. ACM 45,* 9, 50–55.

Schenkel, R., Crecelius, T., Kacimi, M., Michel, S., Neumann, T., Parreira, J. X., and Weikum, G. 2008. Efficient top-k querying over social-tagging networks. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08)*. 523–530.

Speretta, M. and Gauch, S. 2005. Personalized search based on user search histories. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'05)*. 622–628.

Sugiyama, K., Hatano, K., and Yoshikawa, M. 2004. Adaptive web search based on user profile constructed without any effort from users. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*. 675–684.

Tan, G. and Jarvis, S. A. 2007. Improving the fault resilience of overlay multicast for media streaming. *IEEE Trans. Parall. Distrib. Syst. 18*, 721–734.

Teevan, J., Dumais, S. T., and Horvitz, E. 2007. Characterizing the value of personalizing search. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'07)*. 757–758.

Vlachou, A., Doulkeridis, C., Norvag, K., and Vazirgiannis, M. 2008. On efficient top-k query processing in highly distributed environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. 753–764.

Voulgaris, S. and van Steen, M. 2005. Epidemic-Style management of semantic overlays for content-based searching. In *Proceedings of the 11th International European Conference on Parallel and Distributed Computing (Euro-Par'05)*. 1143–1152.

Witten, I. H., Moffat, A., and Bell, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers.

Zegura, E., Calvert, K., and Bhattacharjee, S. 1996. How to model an internetwork. In *Proceedings of the 15th International Conference on Computer Communications (INFOCOM'96)*. 594–602.

Received October 2010; revised April 2011; accepted July 2011