Gossiping Personalized Queries

Xiao Bai, Marin Bertier, Rachid Guerraoui, Anne-Marie Kermarrec, Vincent Leroy

Gossiping Personalized Queries*

Xiao Bai
INSA Rennes, France
xbai@irisa.fr

Marin Bertier
INSA Rennes, France
mbertier@irisa.fr

Rachid Guerraoui
EPFL, Swizerland
rachid.guerraoui@epfl.ch

Anne-Marie Kermarrec
INRIA, Rennes, France
anne-marie.kermarrec@inria.fr

Vincent Leroy
INSA Rennes, France
vleroy@irisa.fr

ABSTRACT

This paper presents P3Q, a fully decentralized gossip-based protocol to personalize query processing in social tagging systems. P3Q dynamically associates each user with social acquaintances sharing similar tagging behaviours. 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 P3Q for top-k query processing. More specifically, we show that on a 10,000-user delicious trace, with little storage at each user, the queries are accurately computed within reasonable time and bandwidth consumption. We also report on the inherent ability of P3Q to cope with users updating profiles and departing.

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 are huge sources of information but enabling their exploration is challenging because of the unstructured nature of tagging and the lack of any fixed ontology. Clearly, the user freedom to choose tags turns out to be 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 behaviours. 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. In contrast, a Keanu Reeves fan may just look for this movie. User affinities can disambiguate these situations.

Several personalized approaches have been proposed to leverage social networks into search procedures [4, 17]. So far, however, these approaches focused mainly on explicit social networks, i.e., 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 you can learn a lot from people you might not know but with whom you share many interests. In this paper, we address the problem of leveraging implicit acquaintances in ranking the results of top-k queries in a large scale collaborative tagging systems.

This personalization of the query processing is challenging in such a setting given the large amount of information that needs to be maintained on a user basis, especially given the dynamic nature of the systems where users continuously join or leave, as well as periodically change their profiles by tagging new items.

Centralized solutions do clearly not scale. Using the centralized approach of [1] to score an item based on the tagging behaviours of users sharing similar preferences leads to maintain one inverted list per (user, tag). This reveals extremely space consuming: under the estimation of [1], maintaining the inverted lists for $100,000$ users would require several terabytes in delicious. Knowing that the actual system has more than 5 millions users and 150 millions URLs, a centralized approach of personalization does not seem realistic. The problem of scalability would be more severe in systems like Youtube and Flickr with videos and pictures.

We argue that leveraging implicit social networks in the search process calls for decentralized solutions. Besides being scalable and able to cope with dynamics, decentralized solutions circumvent the danger of central authorities abusing the information at their disposal, e.g., exploiting the user profiles for commercial purposes, or suffering from denial of service attacks as observed in August 2009 on Facebook, Twitter and LiveJournal at once.

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 the profiles of her acquaintances which will then be massively replicated and hence hard to maintain. Not surprisingly, and as shown by our experiments [3], several hundreds of profiles are needed to return reasonable results (in the sense of [1]) in a system of only $10,000$ users. Maintaining all the 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 whenever a query is to be processed. Clearly, this optimizes the storage and maintenance issues but might induce a large number of messages if the other profiles are consulted at the same time or dramatically increase the latency of query processing if they are consulted one by one. In addition, the profiles of temporarily disconnected users would be unavailable which, in turn, would significantly hamper the accuracy of the query processing.

In this paper, we propose P3Q, a *bimodal, gossip-based solution to personalize the query processing*. P3Q 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 tagging profiles. Every user maintains her social network (a set of IDs) but locally stores a limited subset of profiles, typically those of the most similar users, according to

*This work is supported by the ERC Starting Grant GOSSPLE number 204742

their storage capabilities. The maintenance of the social network is performed in a *lazy* gossip mode, with a fairly low frequency to avoid overloading the network. To limit the bandwidth consumption, users exchange the whole profiles only when these appear to be significantly similar. Instead, the digests of profiles are encoded in Bloom filters and first exchanged to estimate the proximity between profiles.

The very same gossip scheme is also used to process queries with two differences: processing a query is achieved using an *eager* mode of the gossip protocol, i.e., with an increased frequency and the gossip 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. The query is then gossiped further, first to the closest acquaintances and further away according to social proximity, iteratively refining the results. The query contains the list of profiles that should be used to compute the query (based on the social network of the querier) and is computed collaboratively on the fly. Each user reached by the query locally computes her share of the query based on the relevant profiles stored locally and 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 as the time passes, a modified NRA algorithm (No Random Access [11]) is used to retrieve the k most relevant items from the partial results. 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, few gossip cycles are sufficient to compute almost perfect results.

We evaluate P3Q both analytically and experimentally. The analysis assumes a uniform setting of the system parameters and 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 but 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 P3Q using *PeerSim*[13] with a real dataset crawled from delicious with 10,000 users in January 2009. We considered several storage scenarios. We show that even 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 tradeoff between the user's expectation of 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, 90% of the stored information is updated. Meanwhile, P3Q incurs acceptable overload in terms of bandwidth consumption: 13.4K bps are sufficient for maintaining the personal network and 91K bps are sufficient to compute a query. P3Q is also proven robust against user departure: for instance, a massive leaving of 50% users impacts the quality by only 10%.

To summarize, our work is the first to perform search queries in a distributed and personalized manner using implicit user affinities. Whereas we use standard techniques to compute profile proximities (number of common tag-item) and rank queries (NRA), P3Q is generic in this sense and alternative techniques could be used. In fact, our contribution lies in the way we decentralize query processing in a large scale dynamic system and this is not limited to top-k processing: we believe that it could be used in the context of personalized query expansion for example.

The rest of the paper is organized as follows. Section 2 describes our P3Q protocol and analyses its behaviour. Section 3 presents our experimental results. Section 4 concludes our work by discussing related work.

2. THE P3Q 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. $Tagged_{u_i}(i,t)$ captures the fact that user u_i tagged the item i with the tag t. The profile of a user u_i is described as a set of her tagging actions, i.e., $Profile(u_i) = \{Tagged_{u_i}(i,t)\}$. The network is modeled as a directed graph where each node corresponds to a user and an edge presents the link between two users. When there is a directed edge from user u_i to user u_j, u_j is considered as a neighbour of u_i. 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 P3Q, except for her own profile, a user maintains two data structures: a personal network and a random view (Figure 1).

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

$$\begin{aligned} Score_{u_i}(u_j) &= |Profile(u_i) \cap Profile(u_j)| \\ &= |\{(i,t)|Tagged_{u_i}(i,t) \wedge Tagged_{u_j}(i,t)\}| \end{aligned}$$

As a tag can be used for different items and an item may receive different tags from different users, the metric we use takes the users' preferences on both topics and specific objects into account. The higher the score, the more interests are shared between u_i and u_j. In fact, this distance is application-specific and P3Q is independent of the way similarity is defined. The query results of a user u_i only depend on the profiles of the neighbours in 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 neighbours u_j having the highest $Score_{u_i}(u_j)$ 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.

In order to limit the overhead of the protocol, a digest of profile ($Digest(u_j)$) is also stored along with each neighbour in the personal network. The digest is encoded using a Bloom filter [6] and only contains the items tagged by each user. This is used in the gossip protocol to retrieve the full profile of a user if the digest indicates that the user might be a neighbour or her profile should be updated.

Random view. Each user also maintains a set of r neighbours, called a random view, selected uniformly at random from the whole network. These users are used to ensure that the network remains connected [14]. In addition, this enables the discovery of new simi-

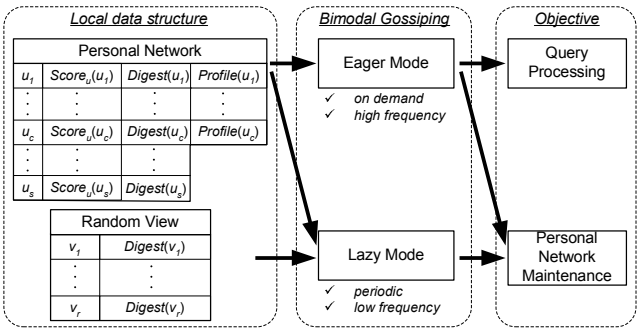


Figure 1: System model

lar neighbours. A digest of profile is also stored for each neighbour.

2.2 Bimodal gossiping

P3Q relies on a two-mode gossip protocol as represented in Figure 1. The *lazy mode* runs *periodically* at a low frequency and is responsible of 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 neighbours in personal networks for collecting the profiles of similar neighbours required to compute the query but which are not stored by the querier.

In short, a generic p2p 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 [14, 26] and information dissemination [10]. We now describe the lazy and eager modes of the P3Q 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 [14], maintains the random view of a user: at each cycle, a user u_i sends the r digests [1] to a neighbour v_j picked uniformly at random from her random view and receives r digests from v_j. Then r digests among the $2r$ digests are randomly selected to form the new random view of u_i. v_j follows the same algorithm.

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

This message is composed of a subset of her neighbours' profiles, randomly selected from the c profiles stored in u_i's personal network. In turn u_j sends back to u_i a subset of her neighbours' profiles. Then u_i computes $Score_{u_i}(u_l)$ for each received user u_l and $Score_{u_i}(v_j)$ for each user v_j in her random view. The profile of v_j is obtained by directly contacting v_j if $Digest(v_j)$ contains at least one item tagged by u_i, which implies that v_j may be a potential neighbour of u_i. User u_i (u_j) keeps in her personal network the s users with the highest (positive) scores and the profiles of the top ranked c users are locally stored.

The bottom layer and the top layer run in parallel, i.e., at each cycle, a user gossips with a neighbour from her random view and a neighbour 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 neighbours that have not been recognized by current neighbours 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. The first exchange (1-15) of the digests enables to approximate the similarity between users using the profile digests: if two users have no common item in their profiles, they simply do not qualify as neighbours in their respective personal networks, or if the users have already in each other's personal network and their profile digests remain the same. In both cases, there is no need to exchange the profiles. The second exchange (16-26) of common items and their associated tags 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 the other neighbours' profiles. The last exchange (27-31) only happens if there are indeed profiles that should be stored.

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 neighbours 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 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. Firstly, it is unlikely that the profiles stored by random neighbours are required by the querier. Secondly, 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 [14].

The eager mode of P3Q works as follows. The querier u_i first processes her query Q based on the profiles in her personal network. This provides a partial and local result to the query. The *remaining list* of a user u_i for query Q, denoted $L_Q(u_i)$, is the set of users from her personal network whose profiles are not stored locally. Those profiles are discovered through gossip. User u_i initiates a gossip with a neighbour u_j having the oldest timestamp in $L_Q(u_i)$ and attaches the query and the remaining list to the gos-

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

Algorithm 1 Gossiping profiles in the top layer

Input: $Profile(u_i)$ & received profile digests
Output: new $Network(u_i)$
for each received $Digest(u_l)$ **do**
  if $u_l \in Network(u_i)$ **then**
    if $Digest(u_l)$ does not change **then**
      drop $Digest(u_l)$
    else
      add u_l to $Candidates$
    end if
  else if u_l has no common item with u_i **then**
    drop $Digest(u_l)$
  else
    add u_l to $Candidates$
  end if
end for
if $Candidates$ is not empty **then**
  for each u_l in $Candidates$ **do**
    require her tagging actions for the common items with u_i
  end for
  receive the required information
  for each u_l in $Candidates$ **do**
    compute $Score_{u_i}(u_l)$
    if $Score_{u_i}(u_l)$ is one of the s highest scores **then**
      add u_l to $Network(u_i)$ with $Score_{u_i}(u_l)$ and $Digest(u_l)$
    end if
  end for
  for each u_l added to $Network(u_i)$ **do**
    if $Score_{u_i}(u_l)$ is one of the c highest scores **then**
      require the rest of the tagging actions in $Profile(u_l)$
    end if
  end for
end if

sip message containing the profile digests she wants to send to u_j as described in the lazy mode. When u_j receives the message, she checks whether she locally stores the profiles of the users in $L_Q(u_i)$, removes them from the remaining list and processes her share of the query locally. This updated remaining list is then split into two parts: a portion α ($0 \leq \alpha \leq 1$) is sent back to the querier in her gossip message containing the profile digests, the remaining portion forms her remaining list for the query Q: $L_Q(u_j)$. 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 her other neighbours. The partial result of the query is sent back to the querier in a message independent on the gossip. A list of users whose profiles are used for the computation are 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, 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.

In the second cycle, both u_i and u_j gossip with one of their neighbours 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 neighbour 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 interested by the querier. Receiving the gossip message, the gossip destinations of u_i and u_j do the same processing as u_j did in the first cycle. At the end of the second cycle, the querier updates the query results again 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 neighbours' profiles are used for the processing.

Algorithm 2 Querier's processing in eager mode

Input: querier u_i's query Q & $Network(u_i)$
Output: personalized query results of Q
process Q with the profiles in $Network(u_i)$
if u_i stores all her neighbours profiles **then**
  display query results and return
else
  build the remaining list $L_Q(u_i)$
  repeat
    gossip with a neighbour u_j in $L_Q(u_i)$
    receive new $L_Q(u_i)$ from u_j
    receive partial results from collaborating users
    compute and display new results with available information
  until all neighbours' profiles are used for query processing
end if

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 parameter 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 P3Q 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 update the personal networks of the users participating in the gossip (Section 3.4.1).

Algorithm 3 Gossiping queries in eager mode

Gossip Initiator (u_{init})
for each cycle **do**
  if $|L_Q(u_{init})| > 0$ **then**
    if $\exists u_l \in L_Q(u_{init})$ & $u_l \in Network(u_{init})$ **then**
      $u_{dest} \leftarrow u_l$ with maximum timestamp
      set u_{dest}'s timestamp to 0
    else
      select u_{dest} from $L_Q(u_{init})$
    end if
    send Q and $L_Q(u_{init})$ to u_{dest} in gossip message
    receive gossip message containing new $L_Q(u_{init})$ from u_{dest}
    maintain personal network as in lazy mode
  end if
end for

Gossip Destination (u_{dest})
loop
  receive gossip message containing Q and $L_Q(u_{init})$ from u_{init}
  remove each u_l from $L_Q(u_{init})$ if $Profile(u_l)$ is stored by u_{dest}
  $L_Q(u_{dest}) \leftarrow (1-\alpha)$*$|L_Q(u_{init})|$ users from $L_Q(u_{init})$
  $L_Q(u_{init}) \leftarrow L_Q(u_{init}) \backslash L_Q(u_{dest})$
  send $L_Q(u_{init})$ to u_{init} in gossip message
  process Q with profiles required by u_{init} and stored by u_{dest}
  send partial result to the querier
  maintain personal network as in lazy mode
end loop

2.3 Collaborative top-k query processing

We illustrate in this section the collaborative query processing in P3Q in the context of top-k processing.

Queries and scoring. We consider a query $Q = \{u_i, t_1, ..., t_n\}$, issued by a user u_i with a set of tags $t_1, ..., t_n$. The personalized top-k processing for Q aims to return the k items having the highest relevance scores from u_i's personal network. More specifically, we define the score of an item i for a user u_j and a query Q as the number of tags in Q used by u_j to annotate i, i.e., $Score_{u_j,Q}(i) = \{t_m | t_m \in Q, Tagged_{u_j}(i, t_m)\}$. We define the relevance score of the item i for the user u_i's query Q as the sum of $Score_{u_j,Q}(i)$ of each neighbour u_j in u_i's personal network, i.e.,

$$Score(Q, i) = \sum_{u_j \in Network(u_i)} Score_{u_j,Q}(i)$$

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

Top-k processing in P3Q. As presented above, in P3Q, 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 P3Q, once a user u_j receives a query Q, she computes a partial result for Q with the profiles that she stores and 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 $GoodProfiles(u_j, Q)$. u_j 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 $Score_{u_l,Q}(i)$ for each $Profile(u_l)$ in $GoodProfiles(u_j, Q)$, i.e.,

$$PartialScore_{u_j}(i) = \sum_{Profile(u_l) \in GoodProfiles(u_j,Q)} Score_{u_l,Q}(i)$$

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 P3Q as the partial result lists in P3Q are computed on the fly and asynchronously provided to the querier. So we adapt the classical NRA (No Random Access)[11] algorithm to P3Q while minimizing the processing time. In P3Q, at the end of each cycle, k items are returned to the querier. Algorithm 4 shows the pseudo-code 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 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.

Algorithm 4 Per cycle top-k processing of the querier

```
1.  Input: u_i's query Q & candidate heap & old partial result lists & new partial result lists
2.  Output: new top-k items
3.  ScanningLists ← new partial result lists
4.  ScanningPosition ← 1
5.  while worst-case score of the kth 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 the item i in the ScanningPosition of l
8.          update the last seen value and last scanned position of l
9.          if i ∈ 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 ← ScanningPosition+1
18.     for each partial result list l ∈ 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
```

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 kth item. Several optimizations are possible to incorporate into the basic algorithm, like not re-ranking the candidate heap once an item is modified, but they are out of the scope of this paper.

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.

Theorem 2.1 *Given a query Q and the querier u_i's remaining list of length L, u_i 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 neighbours. After X profiles are found, the length of her remaining list becomes $\alpha(l - X)$ while her neighbour obtains a remaining list of length $(1 - \alpha)(l - X)$. If $0.5 \leq \alpha \leq 1$,

comparing to her neighbour, 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_i who has the longest remaining list as she first gossips the query Q. From the definition, we know that u_i gets the best results that her personal network can provide when none of the users reached by Q has a remaining list, i.e., the length of u_i's remaining list becomes 0 as she has the longest one. Note the length of u_i's remaining list after the rth cycle as $L(r)$, we have

$$\begin{aligned}
L(1) &= \alpha(L-X),\\
L(2) &= \alpha[L(1)-X] = \alpha^2 L - \alpha^2 X - \alpha X,\\
&\dots\\
L(r) &= \alpha[L(r-1)-X] = \alpha^r L - \alpha^r X - \alpha^{r-1}X - \dots - \alpha X\\
&= \alpha^r L - \sum_{i=1}^{r}\alpha^i X\\
&= \begin{cases} \alpha^r L - \frac{\alpha(1-\alpha^r)}{1-\alpha}X & 0.5 \le \alpha < 1\\ L - rX & \alpha = 1 \end{cases}
\end{aligned}$$

For u_i 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

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

Theorem 2.2 *Given L and X, the number of cycles for the querier u_i to get the best results for her query Q, $R(\alpha)$, is monotonicly increasing with α if $0.5 \le \alpha < 1$ and monotonicly decreasing 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, $R(\alpha)$ is monotonicly increasing 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, $R(\alpha)$ is monotonicly decreasing 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 simultanously, i.e., their remaining lists become 0 at the same cycle. Then at the first cycle, one new user is involved except for the querier. So the total number of involved users is 2. Using mathematical induction, if at the rth 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 gossips with another user, which implies that 2^r new users are involved. So 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 in by them. So $2^{R(\alpha)}$ 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, 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 the 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 first cycle, one gossip is done between the querier and one of her neighbour. Supposing all the users involved in the processing finish their remaining lists at the same time, at the second cycle both the querier and her neighbour gossip with another user. So two gossips are done. This process continues until no user has a remaining list. In fact, at the rth cycle, 2^{r-1} gossips are done. So 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. □

3. EXPERIMENTAL EVALUATION

We first describe in Section 3.1 the delicious dataset and the different scenarios 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 P3Q with respect to storage and bandwidth consumption in Section 3.3. We finally evaluate in Section 3.4 the ability of P3Q to deal with profile changes and user departures.

3.1 Experimental setup

Table 1: Distribution of c

c	10	20	50	100	200	500	1000
$\lambda = 1$	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%

3.1.1 Dataset and query generation

The evaluation of P3Q has been conducted in PeerSim [13], an open source simulator for P2P protocols. The dataset used in the evaluation was crawled in January 2009 from delicious. 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. The distribution of tagging behaviours follows a long tail distribution as most items and tags are used by few users [18]. We reduce the dataset by randomly picking $10,000$ users and building their profiles with the items and tags used by at least 10 distinct users. 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[2]. In the experiments reported below, 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. 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 1000 in our simulations. In fact, regardless of the size of personal network, the querier can get the accurate results within limited cycles (Theorem 2.1). Each user stores c profiles of the most similar neighbours in her personal network. Several values for c are considered in the evaluation. The goal of P3Q 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 capabilities in both storage and bandwidth.

To emphasize the effectiveness of our protocol, we first consider uniform systems where all users have identical storages. We vary the value of c and it is set to 10, 20, 50, 100, 200, 500 or 1000 respectively in 7 different scenarios. Two heterogeneous settings with respect to storage capabilities 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 1. 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 storage space.

3.2 Qualitative P3Q evaluation

3.2.1 Personal network maintenance in lazy mode

[2]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 multi-word 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.

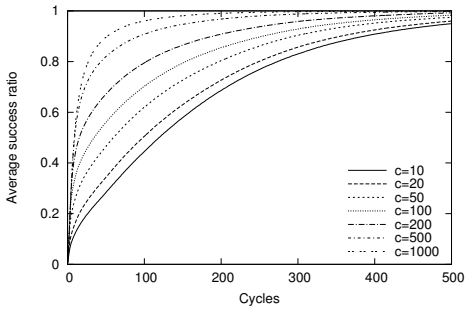


Figure 2: Convergence speed

We first evaluate the ability of P3Q to discover users having similar tagging behaviours. 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 off-line 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 neighbours 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, i.e., $success\_ratio =$

$$\frac{1}{|U|}\sum_{u_i \in U}\frac{Number\ of\ Good\ Neighbours\ In\ Current\ Network}{Number\ of\ Neighbours\ In\ Ideal\ Personal\ Network}$$

$success\_ratio$ is 1 if all users find 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 neighbours for their personal networks (convergence) and the more bandwidth is required. In the evaluation, we set the size of random view to 10, so that at each cycle, 10 profile digests are exchanged in the bottom layer of the gossip protocol (lazy mode). In the top layer, if more than 50 profiles are stored in a user's personal network, 50 random ones among them are exchanged in each cycle. Otherwise, all the profiles are exchanged.

Figure 2 shows the convergence speed assuming uniform storages 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 neighbours more opportunities to discover new neighbours increasing the diversity of profiles proposed in each gossip. Yet, even when only 10 profiles are stored, at the end of the 200th cycle, more than 68% of neighbours in the personal networks are identified. If the users provide sufficient storages, 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 above, the queries are processed through the eager mode of P3Q. 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 P3Q are then compared to this baseline. The *recall* [27] R_k is then measured and computed as follows:

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

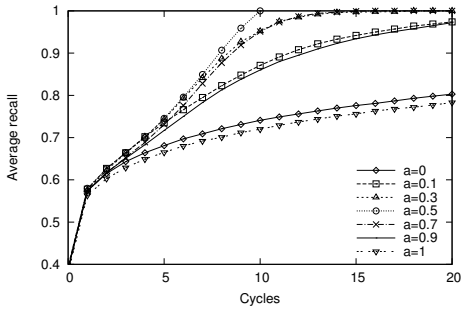


Figure 3: Average recall evolution with different α ($c = 10$)

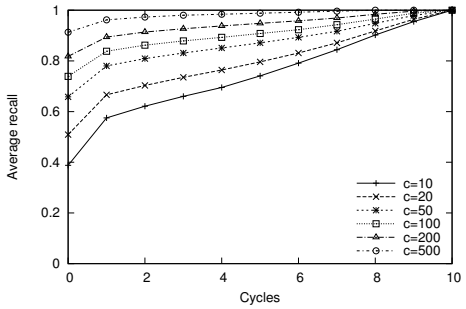


Figure 4: Average recall evolution with different c ($\alpha = 0.5$)

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 depends on the query and the user who generates it. In this context, an ideal $recall = 1$ means that all queries processed in P3Q achieve the same top-k results as the baseline.

Figure 3 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 [4]. In contrast, $\alpha = 1$ means only the neighbours 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 3).

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 3 that the parameter α has an important impact on the top-k processing speed: $\alpha = 0.5$ outperforms other values and the closer α is to 0.5, the faster the top-10 results approach the reference. This confirms our analytical measures.

Figure 4 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, i.e., $R_{10} = 1$. 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 long time and the relatively satisfactory results can be get 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).

3.3 Cost analysis

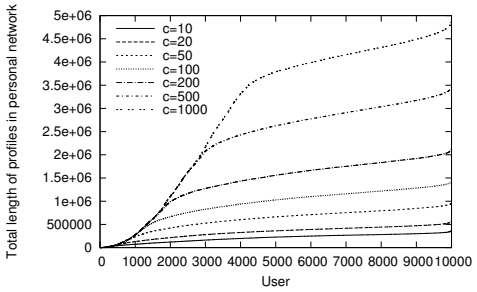


Figure 5: Space requirement

3.3.1 Storage requirements

As opposed to the centralized personalized top-k processing approach presented in [1] where the users store all their neighbours' profiles and the related inverted lists, users in P3Q only store a limited number of their neighbours' profiles significantly limiting the storage requirements.

Each user stores the profile digests of all its neighbours in her personal network and random view and the profiles of c closest neighbours. In our experiments, on average, each user tags 249 items and more than 99% users tag less than 2000 items. Bloom filter of size 20K bits is used to ensure a relative low false positive rate (0.1%). The storage required for the profile digests is constant for all users and is 2.525M bytes in our experiments. 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 [1] 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 5 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 sufficient number of neighbours 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 128 bits MD4 hash value and each user has a 4 bytes ID. Assuming that each tag can be identically presented as a 16 bytes string, a tagging action takes 36 bytes. Storing 10 profiles in the personal network, requires only 12.5MB. This requirement can even be fulfilled by mobile devices with limited capabilities.

3.3.2 Bandwidth consumption

Due to the periodic behaviour of lazy gossiping and the burst 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 $\lambda = 4$.

Personal network maintenance traffic. As mentioned above, 50 profile digests are regularly transferred by each user having more than 50 profiles in her personal network. This imposes a transmission of 125K bytes for each user. Only 25K or $50K$ bytes are transmitted for users having 10 or 20 profiles in their personal

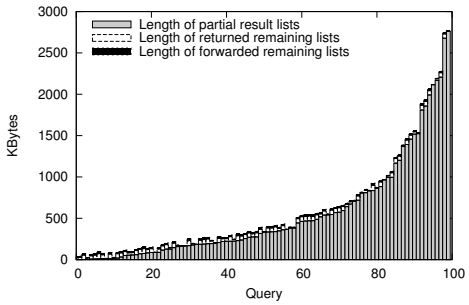


Figure 6: Bandwidth for query processing ($\lambda = 1$)

networks.

Except for the profile digests, the information received by each user for maintaining her personal network consists of two parts: *(i)* the common items and the associated tags to compute the similarity scores 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, 87.6% of users in the system have to transmit further information for measuring the similarity while only 4.1% of them require the exchange of the whole profiles. For these users, 15.85K bytes and 503K bytes are transmitted respectively if they have such need in a certain cycle. Practically, the maximum information transmitted in a single cycle does not exceed 5M bytes. Similar performance is observed in the $\lambda = 4$ scenario . On average, at each cycle, 88.0% users have to transmit 23.68K bytes for measuring the similarity while 12.95% of them need the whole profiles of 545K bytes. This is due to the fact that more neighbours 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 capability. In the bottom layer of the lazy gossip, 10 profile digests of 25K bytes are exchanged at each cycle.

Query processing traffic. When a query is gossiped in the system, 3 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 bytes ID. The score of each item in the partial result list can also be presented by a 4 bytes integer. Figure 6 depicts the quantity of information transmitted in the scenario with $\lambda = 1$ to answer a query. For the visibility of the figure, only 100 queries are randomly picked from all the queries and shown in it. 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 comparing to other information. The value on the X-axis represents an individual query.

On average, in the $\lambda = 1$ scenario 573K bytes are transmitted to answer a query and in the $\lambda = 4$ scenario, 360K bytes are transmitted. The reason is that in a system where many users have large storages, 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.

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, 228 such messages are transferred to the querier in the $\lambda = 1$ scenario and 70 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 Dynamism

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 the users are not online all the time. We evaluate in this section the impact of both forms of dynamics, respectively the profile dynamics and the churn on the same delicious trace.

3.4.1 Profile dynamism

First, we analyze the underlying patterns of changes in the system during the whole year of 2008. We observe that every week more than 3000 users change their profiles while less than 60 new users are involved in the system. As new tagging behaviours dominate new users, we focus on the impact of changes in 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 2. 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, i.e., each user adds the new tagging actions happened in 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 P3Q 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, i.e., $AUR =$

$$\frac{1}{|U|}\sum_{u_i \in U} \frac{Number\ of\ Updated\ Profiles\ In\ Network(u_i)}{Number\ of\ Profiles\ In\ Network(u_i)\ Owing\ Update}$$

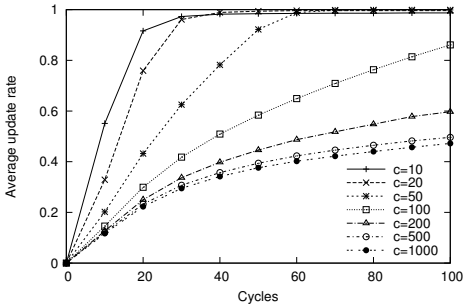
AUR attains 1 when the profiles in all users' personal networks are up-to-date.

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 having the same number of profiles (c) in their personal networks. We consider the day where 1540 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 2 summarizes the influence of profile changes in different settings.

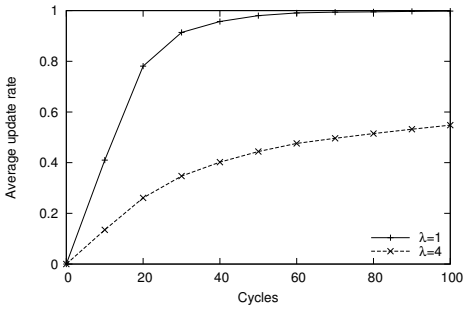
In lazy mode, i.e. after users changing their profiles, no query is generated, we compute the average update rate after each cycle. This is illustrated on Figure 7(a). We observe that a small number of stored profiles (c) guarantee a high average update rate. After 30 cycles, more than 95% profiles are updated in both systems for users storing 10 or 20 profiles while only 40% of the profile are updated after 100 cycles for the users storing 500 or 1000 profiles.

Table 2: Influence of profile changes in different systems

c	% 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
1000	88.2%	105	388



(a) Uniform c



(b) Different distributions of c

Figure 7: AUR evolution in lazy mode

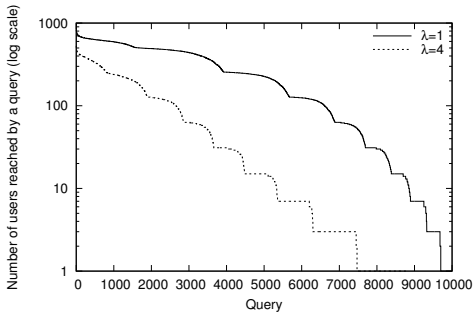


Figure 8: Number of users reached by the query

Not surprisingly, the more profiles are stored in the personal network, the more difficult it is to keep all of them up-to-date.

We now consider in the following the heterogeneous scenarios with $\lambda = 1$ and $\lambda = 4$ for further analysis. Figure 7(b) confirms the former observation that if most of the users in the system have small number of stored profiles, it is easier to keep them up-to-date.

We now consider the impact of running the eager mode on the

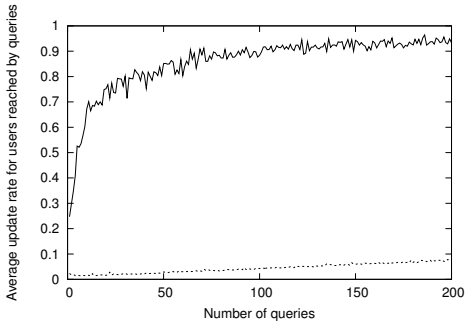


Figure 9: AUR evolution in eager mode

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, i.e., the small fraction of users reached by the query. This has a significant impact of the freshness of the personal networks of such users. To illustrate the ability of the eager mode to cope very 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 2 heterogeneous scenarios is shown in Figure 8. The X-axis can be considered as query identification and the queries are ranked in descending order of their Y-axis values. On average, during the processing of a single query, 256 users are reached by the query in the $\lambda = 1$ scenario while 75 users are reached in the $\lambda = 4$ scenario.

Figure 9 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. We observe that if most users have small storages ($\lambda = 1$), the acceleration effect of eager gossip is prominent. After answering a single query, on average, about 24% profiles are updated. 10 consecutive queries enable all the users reached by the queries to update more than 60% 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 storages. Moreover, with small storage, in each cycle of gossip the same profiles are proposed. Once a profile is updated, gossip protocol ensures a fast dissemination.

Updating neighbours. Active tagging behaviours of users may not only impact on the stored profiles, but also impact the personal networks themselves. Considering the same day in the simulation, we observe that the changes in profiles led 1719 users to change an average of 2 (maximum 148) neighbours 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 neighbours 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 neighbours are discovered, the ratio is still 0 unless her personal network is completed.

Figure 10 shows that, in both settings, after 30 cycles, half of the users have discovered all their relevant neighbours and at the 100th cycle, the number reaches 80%. This illustrates the fact that users efficiently capture the new trends in their personal networks. We do not display the results for the eager mode, as the eager mode does not impact the neighbours discovery as the gossip operations are limited to the querier's neighbourhood.

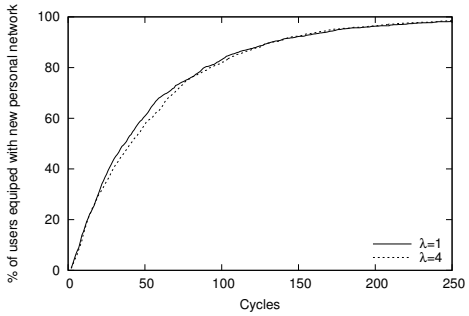


Figure 10: Personal network evolution in lazy mode

3.4.2 Churn

Users who do not store all their neighbours' profiles should collect more information through gossiping. However, the original owner of a profile may not be online at query time. We now evaluate the failure-resilience capability of P3Q. 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 has left, the replicas of her profile would not be out-of-date because her opinion on the tagged items keeps meaningful and no new tagging actions can be added 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 11 illustrates the impact of the number of leaving users on the top-k processing in the $\lambda = 1$ and $\lambda = 4$ scenarios respectively. p is the percentage of leaving users. Obviously, the more users leave the system, the slower the average recall improves along time. However, even 90% users have left, at the end of the 10th cycle, on average, about 8 relevant items can be returned to the querier in the $\lambda = 1$ scenario (Figure 11(a)). Better results are observed in the $\lambda = 4$ scenario (Figure 11(b)). This is due to the fact that in the latter system, more replicas are available thanks to larger storages of the remaining users. If only 10% of users leave, the degradation on processing time is very small. Yet, the average recall fails to get 1 no matter how long the users wait because a certain number of queriers can not find all the profiles in their personal networks (Figure 11(c)). However, even if 50% of users leave simultaneously in the $\lambda = 4$ scenario, the percentage of non complete queries remains smaller than 5%. Yet, those results confirm that our system is robust in 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 Summary

Our evaluations demonstrate that the users get good results immediately and can be further satisfied with accurate results within a small number of gossip cycles. Although P3Q takes more time to build the personal networks if the users store less, once most of the neighbours are identified, P3Q guarantees a better freshness of the local stored information and consumes less bandwidth for the personal network maintenance. However, users still have the possibility to store more information if they are willing to get a better result immediately. Consider for instance the scenario with $\lambda = 1$. Assume 1 minute per cycle and 5 seconds per cycle are used in the lazy mode and the eager mode respectively, the query can be accurately answered within 50 seconds with an average bandwidth consumption of 91K bps (bits per second) for the querier. The background traffic for maintaining the personal network through lazy gossip is only 13.4K bps and this may increase to 121K bps in eager gossip. Even if all users simultaneously change their profiles, in half an hour, 90% of the local stored information is updated and more than 50% users' new neighbours are identified.

4. CONCLUDING REMARKS

Early in 2000, [15] pointed out the importance for future search engines to leverage (either explicit or implicit) context information to improve the search process. This was also confirmed in [24] where personalized search was considered as a promising way for boosting the quality of search engines.

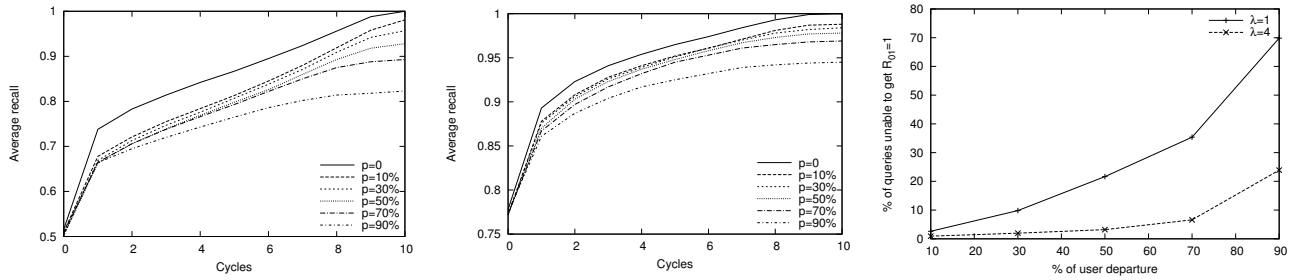
Two general approaches for search personalization were described in [20]: query expansion and result processing. The first approach tries to append new terms to a query in order to better reflect the user's profile ([7, 8]). The second approach runs the original query but re-ranks the returned results based on the user's profile. A wide range of user activities have also been considered to enhance re-ranking, including user's query histories [22], browsing histories [23] and tagging behaviours [19].

Various community-aware ranking algorithms have been developed to explore the relationships between users for personalizing information retrieval. A social scoring function, leveraging the strength of user relations and correlations among different tags, was proposed in [21] to improve the top-k quality. Various notions of user affinity and social relations were discussed in [2, 21]. A general indexing and query processing framework encompassing a wide class of scoring functions and networks was developed in [1]. 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 shown that building the inverted lists for each (user, tag) pair is too space-intensive, while clustering users with similar tagging behaviours and building inverted lists for each cluster impacts the processing time.

The way P3Q performs the top-k processing is inspired by the network-aware search technique of [1]. P3Q is however decentralized, in terms of both storage and processing, and this we believe is the key to its scalability. Some approaches to decentralize top-k processing have been proposed. In [16], pre-computed inverted lists are distributed across nodes and partial information is transmitted in the network progressively to approximate top-k results. In [5], a Chord-based DHT is used to partition the term space and each node is responsible for a random set of terms. The query is then routed to the nodes responsible of the query related terms. These approaches differ from P3Q which does not rely on any global dictionary or specific mechanism to organize the data in the system.

SPEERTO [25] explores a skyline-based routing which forwards the top-k queries among super-nodes to minimize the data transferred in the system. PlanetP [9] uses gossip to globally replicate a membership directory and a term-to-peer context index. A searching node first identifies the set of nodes having the query related terms with the global index and then ranks the relevant documents (returned by these nodes) to determine the most pertinent ones. Unlike in P3Q, none of these approaches achieves personalization.

In the context of top-k processing, explicit (declared) social connections have also been considered. In [17] the potential for using social networks to enhance Internet search is discussed. 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



(a) Average recall evolution ($\lambda = 1$) (b) Average recall evolution ($\lambda = 4$) (c) Queries unable to get $R_{10} = 1$

Figure 11: Impact of user departure on top-k

P3Q user with a pre-defined explicit network (e.g., explicit social network in Facebook) as input would be straightforward: only the eager mode of P3Q would suffice.

P3Q relies on a gossip-based protocol to discover and leverage implicit relations 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 [10], aggregation [12] and overlay topology management [26].

To summarize, and to the best of knowledge, our work is the first to perform search queries in a distributed and personalized manner using implicit user affinities.

5. REFERENCES

[1] S. Amer-Yahia, M. Benedikt, V. Lakshmanan, and J. Stoyanovic. Efficient network aware search in collaborative tagging sites. In *VLDB'08*.

[2] S. Amer-Yahia, C. Marlow, C. Yu, and J. Stoyanovich. Leveraging tagging to model user interests in del.icio.us. In *AAAI-SIP'08*.

[3] X. Bai, M. Bertier, R. Guerraoui, and A. Kermarrec. Personalized top-k processing in peer-to-peer system. In *SNS'09*.

[4] M. Bender, T. Crecelius, M. Kacimi, S. Michel, J. Parreira, and G. Weikum. Peer-to-peer information search: Semantic, social, or spiritual? *IEEE Data Eng. Bull.'07*.

[5] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. Minerva: collaborative p2p search. In *VLDB '05*, pages 1263–1266. VLDB Endowment.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, 1970.

[7] M. Carman, M.Baillie, and F. Crestani. Tag data and personalized information retrieval. In *SSM'08*.

[8] P. Chirita, C. Firan, and W. Nejdl. Personalized query expansion for the web. In *SIGIR'07*.

[9] F. Cuenca-Acuna, C. Peery, R. Martin, and T. Nguyen. Planetp: using gossiping to build content addressable peer-to-peer information sharing communities. In *HPDC'03*.

[10] P. T. Eugster, R. Guerraoui, A. M. Kermarrec, and L. Massoulie. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, 2004.

[11] R. Fagin. Combining fuzzy information: an overview. In *SIGMOD'02*.

[12] M. Jelasity. An approach to massively distributed aggregate computing on peer-to-peer networks. In *PDP'04*.

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

[14] M. Jelasity, S. Voulgaris, R. Guerraoui, A. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3):8, 2007.

[15] S. Lawrence. Context in web search. *IEEE Data Engineering Bulletin*, 23:25–32, 2000.

[16] S. Michel, P. Triantafillou, and G. Weikum. Klee: a framework for distributed top-k query algorithms. In *VLDB '05*.

[17] A. Mislove, K. Gummadi, and P. Druschel. Exploiting social networks for internet search. In *HotNets'06*.

[18] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC '07*.

[19] M. Noll and C. Meinel. Web search personalization via social bookmarking and tagging. In *ASWC'07*.

[20] J. Pitkow, H. Schütze, T. Cass, R. Cooley, D. Turnbull, A. Edmonds, E. Adar, and T. Breuel. Personalized search. *Commun. ACM*, 45(9):50–55, 2002.

[21] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *SIGIR '08*.

[22] M. Speretta and S. Gauch. Personalized search based on user search histories. In *WI'05*.

[23] K. Sugiyama, K. Hatano, and M. Yoshikawa. Adaptive web search based on user profile constructed without any effort from users. In *WWW'04*.

[24] J. Teevan, S. Dumais, and E. Horvitz. Characterizing the value of personalizing search. In *SIGIR '07*.

[25] A. Vlachou, C. Doulkeridis, K. Norvag, and M. Vazirgiannis. On efficient top-k query processing in highly distributed environments. In *SIGMOD '08*.

[26] S. Voulgaris and M. van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Euro-Par'05*.

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