Personalized top-k processing: from centralized to decentralized systems

Xiao Bai

Thèse

THESE INSA Rennes
sous le sceau de l'Université européenne de Bretagne
pour obtenir le titre de
DOCTEUR DE L'INSA DE RENNES
Spécialité : Informatique

présentée par
Xiao BAI
ECOLE DOCTORALE
MATISSE

Personalized top-*k* processing: from centralized to decentralized systems

Thèse soutenue le 08.12.2010
devant le jury composé de :

Rachid Guerraoui
Professor, EPFL / *président*
Sihem Amer-Yahia
Senior Research Scientist, Yahoo! Research NYC / *rapporteur*
Esther Pacitti
Professeur à Université de Montpellier 2 / *rapporteur*
Michel Raynal
Professeur à Université de Rennes 1 / *examinateur*
Anne-Marie Kermarrec
Directrice de Recherche à INRIA Rennes / Directeur de thèse
Marin Bertier
Maître de conférences à INSA de Rennes / Co-directeur de thèse

Personalized top-k processing: from centralized to decentralized systems

Xiao BAI

En partenariat avec

CONTENTS

INTRODUCTION

The prevalence of social networks, blogs, wikis, media-sharing sites and folksonomies in the past few years heralds the arrival of the Web 2.0 era. The Web 2.0 sites provide users the freedom to share their information and express their feelings by posting blogs, submitting comments and tagging content. This differs from the traditional "Web-as-information-source" websites in the sense that users are not only limited to passively browse the content created for them but are also encouraged to contribute to the information creation. The Web 2.0 revolution is transforming the Internet to a collaborative media where users can meet, read and write.

The user-generated content constitutes a rapidly increasing proportion for the Web. In the most famous social networking site *Facebook*, more than 30 billion pieces of content, such as web links, news stories, blog posts, photo albums, etc., are shared each month [1]. In the recently popular mini-blog service *Twitter*, about 750 tweets are generated per second, which account for 65 millions tweets per day [2]. In the meantime, 24 hours of new videos are uploaded to the *YouTube* site every minute [3].

Such user-generated content is thus becoming a richer and richer source of information. In 2006, the power of users and their generated content was admitted in *TIME* magazine. The author of the cover story explained in *"Person of the year – You"* [4] that

> *"It's a story about community and collaboration on a scale never seen before. It's about the cosmic compendium of knowledge Wikipedia and the million-channel people's network YouTube and the online metropolis MySpace. It's about the many wresting power from the few and helping one another for nothing and how that will not only change the world, but also change the way the world changes."*

A more convincing example of the power of user-generated content as an active information source was presented after the crash of a *USAirways* flight in New York's Hudson River in January 2009. The first tweet broke that news 4 minutes after the incident, which was around 15 minutes earlier than the mainstream media [5].

However, efficiently and accurately discovering valuable information from the huge amount of information is not an easy task. During the past few years, with the explosion of information, people rely more and more on search engines to access the information in the Internet: 3 billions, 280 millions and 80 millions queries are tackled each day by *Google*, *Yahoo* and *Bing* respectively [6]. Yet, the prevalence of the Web 2.0 applications brings new challenges to traditional search. As previously stated, content is being generated at an unprecedented rate, which makes it more and more difficult and expensive to keep up with the pace of its generation and evolution.

More importantly, users do not have any limit in the content they submit and the way they interpret the content. Such arbitrariness and subjectivity impose higher requirements for search engines to

distinguish "I just watched a plane crash into the Hudson" from "I just watched two swans swim in the Hudson" and bubble the valuable information up. In fact, according to the study of US market research firm *Pear Analytics*, 40% of the messages sent via *Twitter* are "pointless babble" [7].

Moreover, the value of the same information might vary across users. The success of infinite-inventory retailers like *Amazon* and *Netflix* ascribes the long-tail phenomenon [8] and confirms the diversity of user preferences. Personalizing the search for each individual user has in fact attracted a lot of attention from both industry and academe in the past few years. Yet, the emergence of the Web 2.0 applications opens up a new path to understand the user preferences. With Web 2.0, users have the opportunity to explicitly express what they like and what they do not like. Better personalization can be achieved should such information be taken into account while leaning the user preferences.

All these challenges are calling for more technologies that can be leveraged to implement the search in order to make full use of the emerging source of valuable and timely information, while preserving the reliability and the low cost.

Peer-to-peer network is well known for its scalability and robustness. Comparing to the Web 2.0 technology, the peer-to-peer technology gained its popularity several years in advance. In 1999, *Napster* became the major on-line music sharing system. Although the original service was shut down in 2001 for copyright reasons, it paved the way for the success of numerous peer-to-peer applications. The most commonly known applications are for content delivery. Since 2004, the peer-to-peer file sharing applications, like *Gnutella* and *BitTorrant*, are the largest contributors of network traffic on the Internet. Live streaming is another successful application of content delivery in peer-to-peer systems. *PPStream*, one of the most popular peer-to-peer streaming video software, has more than 400 millions installations with 11 millions daily and 100 millions monthly active users [9]. The well-known *Skype* is one of the most widely used Internet phone applications using peer-to-peer technology. A report from *TeleGeography Research* stated that Skype-to-Skype calls accounted for 13% of all international call minutes in 2009 [10].

A peer-to-peer network is a distributed architecture composed of participants that collaborate with each other to share resources, such as processing power, storage and bandwidth, without any centralized coordination server. Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model where the servers supply and the clients consume. As the peers also play as resource suppliers, even if the arrival of new peers increases the demand on the system, the total capacity of the system increases accordingly, which provides the system more scalability. The distributed nature of peer-to-peer networks also improves the robustness of the system by avoiding the single point of failure faced by centralized systems. More importantly, it provides a promising possibility to manage the user-generated content in Web 2.0 applications. As John Robb wrote [11]:

> *"What is Web* 2.0*? It is a system that breaks with the old model of centralized Web sites and moves the power of the Web/ Internet to the desktop. ... Basically, Web* 2.0 *puts the power of the Internet in the hands of the desktop PC user where it belongs."*

The Web 2.0 technology offers users the freedom to create information while the peer-to-peer systems rely on users to deliver information. Both of them are trying to break through the limit of central providers and explore the potential of users as much as possible. Building the Web 2.0 applications on top of the peer-to-peer paradigm could open the door to many interesting innovations. User experience can be further enhanced should users share and search the information created by

themselves in a peer-to-peer way to avoid the probably redundant collecting and re-distributing procedure. This would be the key for efficiency, scalability and robustness.

Contribution Motivated by these observations, we aim to provide efficient search for the Web 2.0 applications. We highlight in this thesis the potential of using user-generated tags to personalize the search and the applicability of such personalized search in fully decentralized peer-to-peer environments. In this work, we focus on the collaborative tagging systems, where users add metadata in the form of tags to describe and share content. However, the idea and the resulting algorithms are not limited to collaborative tagging systems. Due to the intrinsic similarities, these can be easily extended to other Web 2.0 applications compatible with social tagging to support personalized search.

More specifically, we consider the tagging behaviors of users as the indicator of their preferences and associate each user with a set of other users sharing similar interests with her to personalize the search. Given a query, we aim to efficiently find the top-k answers that match the individual needs of the querier. Similar users can be identified either off-line or on-line according to the personalization requirements. We first study the feasibility of a state-of-the-art off-line personalization in peer-to-peer systems. Then we propose a hybrid interest model requiring on-line personalization to improve the search efficiency. Therefore, we examine how this on-line personalization can be integrated to existing centralized collaborative tagging systems. Upon that, we investigate another algorithm that supports on-line personalization in peer-to-peer environments. We detail the contributions of this thesis following the order of the chapters:

Chapter2: Off-line personalized query processing

- P3K is our first protocol that personalizes the top-k processing in peer-to-peer systems using implicit user affinities underlying the tagging behaviors. The main objective of P3K is to show the applicability of a previous personalized approach [12], designed for centralized systems, in fully decentralized environments. P3K relies on a gossip-based protocol to associate each user with a set of social acquaintances. Relevant personalized information is locally maintained to enable efficient top-k processing. Experimental results based on real dataset show that little storage at each user suffices to get almost the same results as the state-of-the-art centralized solution with infinite storage.

- P4Q is an extension of P3K that enhances the system performance in terms of storage, bandwidth and robustness. Queries are gossiped among social acquaintances, computed on the fly in a collaborative, yet partitioned manner, and results are iteratively refined and returned to the querier. P4Q relies on profile digests encoded in Bloom filters to limit the bandwidth during the gossip and actively react to system dynamics. 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.

Chapter3: On-line personalized query processing

- DT^2 aims to personalize the top-k processing of queries reflecting emerging interests of the users. To this end, a hybrid interest model is proposed to leverage both the tagging profile and the current query to personalize the query processing on-line. DT^2 is implemented in

a centralized collaborative tagging system by doing top-k twice: the first top-k associates each user, at query time, a set of appropriate users and the second top-k processes the query with these users. The users are folded into the query processing in an incremental manner to guarantee the balance between result quality and search efficiency. The advantages of on-line personalization are highlighted through experimental evaluations on real datasets.

- DT^2P^2 is a protocol that performs on-line personalization in peer-to-peer systems. Different from the state-of-the-art approaches that perform query processing in peer-to-peer systems through query routing, the DT^2P^2 user who sends the query dynamically maintains a network of users matching her hybrid interests. The query is then iteratively processed in the refined network. In this way, the querier always has full control of the information used for her query processing. This is achieved through a gossip-based protocol. A cache of such networks are employed to enhance the efficiency of query processing.

CHAPTER 1

BACKGROUND

1.1 Context

1.1.1 Collaborative tagging systems

Collaborative tagging has blossomed with Web 2.0 and is an important feature of many Web 2.0 services. This represents the practice of users to freely create and manage tags for annotating content, which helps users remember and organize information, such as email (*Gmail*), web sites (*delicious*), photos (*Flickr*), videos (*Youtube*), blogs (*Technorati*) and academic papers (*CiteULike*). Collaborative tagging system is a typical application of Web 2.0 that specifically supports tagging.

Figure 1.1 gives an example of the popular social bookmarking web service *delicious* [13], which is also the most famous representative of collaborative tagging systems. This figure depicts the URLs bookmarked by the user *John*. Each URL is associated with a set of tags given by *John*, reflecting his understanding of the content linked by this URL. For example, the URL *'html5boilerplate.com'* is tagged with *"bestpractice"*, *"boilerplate"* and *"markup"* by *John*, where *"boilerplate"* and *"markup"* characterize its content and *"bestpractice"* represents *John*'s assessment on its value. These tags can later be used by either *John* or other users to search for this URL.

A critical characteristic of collaborative tagging that promotes social navigation is its prolific vocabulary contributed by the users. Users do not have any limit in the tags they use to describe a resource. People can even invent their personally meaningful tags to facilitate the resource organization. Different users may use *"template"*, *"framework"* or some other synonyms instead of *"boilerplate"* to tag the URL *"html5boilerplate.com"* as *John* did. This enables the most diverse criteria to be allocated to the resources and in this way guarantees a much broader access to them. According to the analysis of [14], only 50% bookmarked URLs in *delicious* contain in their page texts the tags used to annotate them. This reveals that using the user generated tags is a good complement of the full text search. For pictures and videos which do not contain full text themselves, user generated tags provide more comprehensive description that can be leveraged in the search

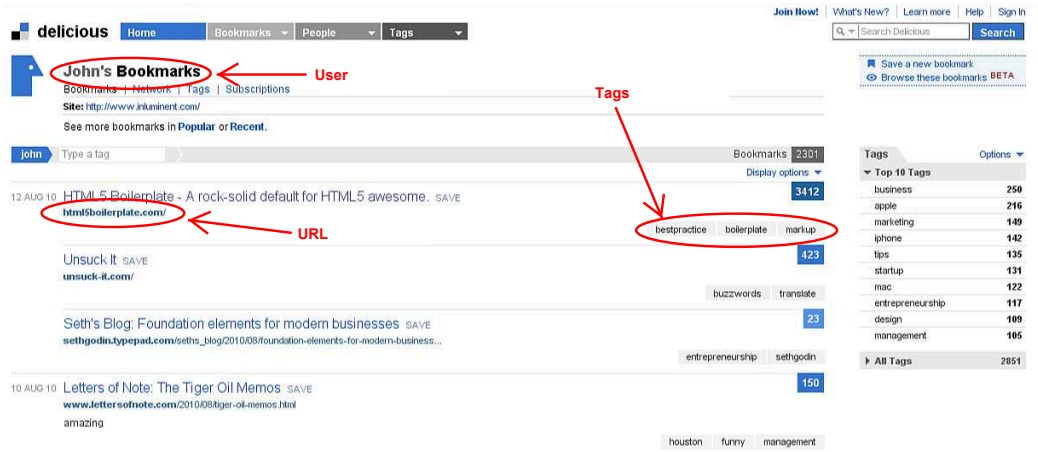


Figure 1.1: Example of *delicious*

procedure, which would have a large impact on improving the search quality.

Yet, performing effective search using the user generated tags is challenging, especially when users search for the most appropriate (top-k) answers that match a potentially ambiguous query.

First, the low participation barrier of tagging and the lack of any pre-defined ontology may result in very noisy description and make the search ambiguous. The URL *"html5boilerplate.com"* judged as *"bestpractice"* by John may be considered *"useless"* by another user. As a result, like all the Web 2.0 applications, it is important to identify valuable information from the huge amount of interfering data generated by the users.

Secondly, collaborative tagging systems confront the rapidly increasing contents as all the Web 2.0 applications. *delicious* claimed more than 5.3 millions users and 180 millions unique bookmarked URLs by the end of 2008 [15]. *CiteULike* [16], the first service for bookmarking and discovering academic papers, indexes more than 4 millions papers and thousands of new papers are added each day [16]. The most popular photo sharing website $Flickr$ [17] had an average inflow of $920,000$ photos in 2006 [18], accumulated to more than 4 billions in October 2009 [19].

In addition, despite of the continuous explosion of resources in collaborative tagging systems, according to the analysis on *delicious*, the tagging vocabulary gradually saturates [20] and the use of tags stabilizes to a power-law distribution [21]. As a result, a tag that is applied to a large number of resources is becoming less descriptive for any given URL in *delicious*, which makes it more challenging to use as a navigational aid. A URL tagged as *"bestpractice"* can be either a recipe for chocolate cake or a tutorial for HTML5 boilerplate. It is thus crucial to bubble up the tutorial for HTML5 boilerplate among the large amount of URLs considered as *"bestpractice"* by different users when *John* searches for the best practice of HTML5 boilerplate.

Personalization is an appealing way in this context to take these challenges up by limiting the search space within a subset of relevant information, which can in turn disambiguate the query processing. We thus carry out our work in the context of collaborative tagging systems and focus in this thesis how the efficient search can be achieved in such systems by personalizing the query processing with user generated tags.

In fact, collaborative tagging has the most typical characteristics of Web 2.0 applications in terms

of user participation, dynamic content and collective intelligence. In the meantime, the search challenges faced by collaborative tagging systems can be considered as microcosms of those in general Web 2.0 applications. The algorithms investigated in this thesis can be easily extended to other Web 2.0 applications that support collaborative tagging.

1.1.2 Large-scale distributed systems

The first challenge faced by collaborative tagging systems is the large scale and the rapid increasing of its information space. Storing the huge amount of information generated by the users is not an easy task for central servers, especially when fine-grained information on a user basis is also to be maintained to provide personalized search. Moreover, any single point of failure of the central server may lead the search service unavailable.

We thus aim to provide *personalized search in large-scale peer-to-peer environments*, where each user is associated with one underlying machine, corresponding to one peer in collaborative tagging systems. There is no central server in the system: each user is in charge of a small portion of the overall information and users collaborate with each other to process the queries. As a result, the capability of the system becomes proportional to its size: the more users are in the system, the larger the storage is and the more queries can be served at the same time.

The main problems we face to personalize the query processing in peer-to-peer systems are *(i)* how to effectively organize the information in the systems; *(ii)* how to efficiently use this information for query processing.

Ideally, a large-scale distributed system should be self-organized and load balancing. The content shared and tagged by users should be evenly distributed across peers in a way that the necessary information can be intentionally accessed at query time.

The system should also adapt fast to changes in very dynamics environments. In the presence of large-scale failure, i.e., a bunch of users leave the system simultaneously, the system should show its self-healing ability in the sense of keeping the network connected and guaranteeing the quantity of available information to perform high quality search. In addition to the active joining and leaving of users in the system, users continuously create, share and tag new content. Such information should be timely incorporated to the search procedure for improving the result quality.

Considering the size of the current collaborative tagging systems and the potentially larger size in the future, the peer-to-peer systems that perform the search in collaborative tagging systems should be highly scalable. It should be able to serve more queries at the same time with the increase of the number of users. Scalability also requires simplicity. The underlying mechanism to organize the information and process the query should be as simple as possible. Given the freedom of each user, a system of millions of users can easily get out of control.

In the next section, we introduce the major techniques concerned in this thesis for self-organizing the information and processing the query that meet the above requirements.

1.2 Major techniques

1.2.1 Gossip-based communication

In order to meet the requirements of simplicity, scalability and robustness of the query processing in peer-to-peer systems, gossip-based protocols appear to be a pragmatic solution. Gossip-based

protocols, also known as epidemic protocols, are main representatives of fully distributed and self-organizing systems.

Since the first use in propagating updates in loosely replicated database to maintain the consistency of replicas in 1987 [22], gossip-based protocols gained their reputation for information dissemination [23, 24, 25], topology management [26, 27, 28], data aggregation [29, 30, 31], failure detection [32, 33], garbage collection [34], load balancing [35] and synchronization [36] in the past twenty years.

The principal operation of a gossip-based protocol is that each peer periodically exchanges the information it possesses with another peer in the network and updates its local information accordingly. The time period is referred to as a *cycle*. Each peer running the gossip-based protocol has two threads: an *active* thread initiating communication with other peers, and a *passive* thread waiting for incoming messages. A gossip between two users can be fully characterized by three functions:

- *PeerSelection:* This function is only performed by the peer running the active thread. The peer selects another peer from the peers it knows to gossip with. This peer can be selected either randomly or based on some specific criteria.
- *DataExchange:* This function specifies which data a peer would send to the peer gossiping with it. It can be performed by the peer running either active or passive thread.
- *DataProcessing:* This function defines, upon receiving the new data from another peer, how to process these data. It can also be performed by both peers depending on the presence of new data.

Figure 1.2 illustrates a gossip between two peers at a given a cycle. We use p to denote the peer who initiates the gossip (active thread) and p' to denote the peer who reacts to the gossip (passive thread).

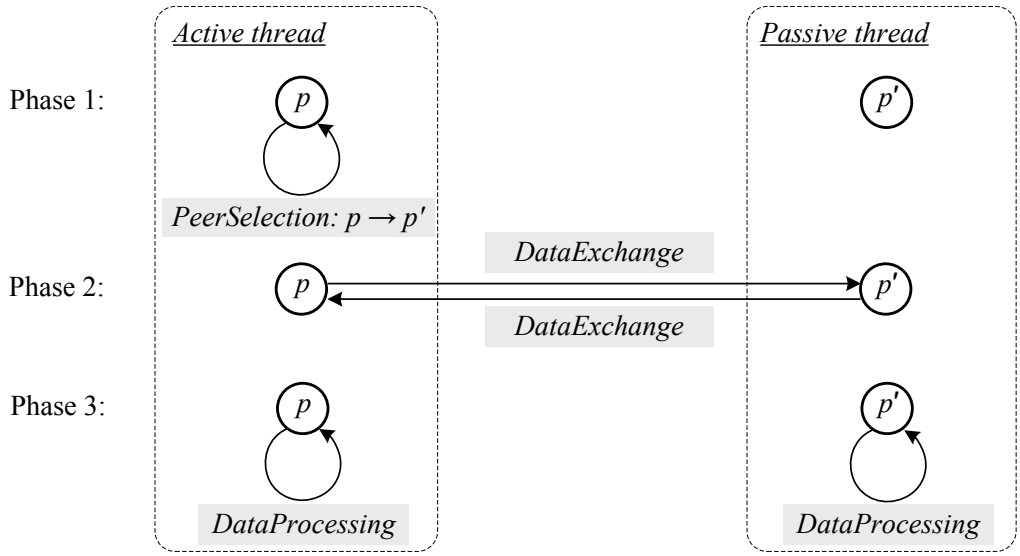


Figure 1.2: A gossip operation between two peers

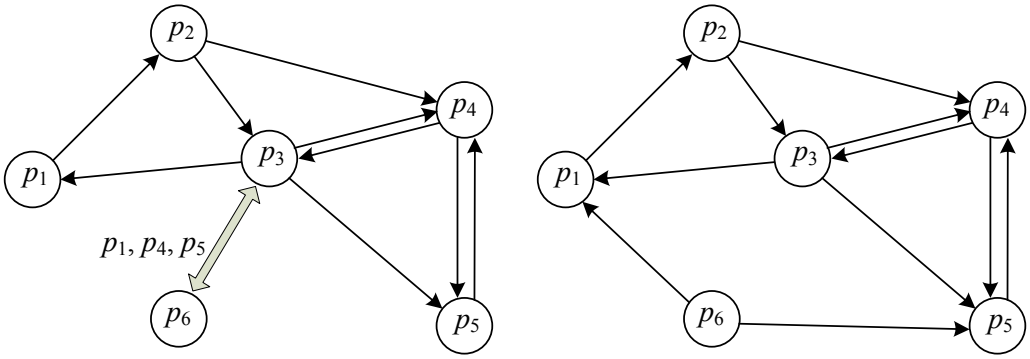
This simple paradigm allows fast spreading the information known by a single peer over the network. The information locally stored by each peer can also be updated through the *DataProcessing* function. According to the application that a gossip-based protocol is dedicated to, these three functions are implemented differently to reflect the desired characteristics. In this thesis, we derive from the general gossiping model mainly the aspects of topology management and information dissemination, which can be leveraged to enable efficient search in peer-to-peer systems.

Topology management Maintaining complete membership table that contains all the peers in a system on each peer is infeasible for large-scale and highly dynamic networks. Tremendous synchronization problems would arise with such effort, especially when a large percentage of peers join in a few minutes [37, 38, 39].

Gossip-based topology management takes this challenge up based on the assumption that each peer only maintains a very small number of neighbors, constituting its *partial view* of the complete network. Through continuously exchanging their neighbors and refreshing their partial views, peers can self-organize into different topologies that suit certain applications.

More specifically, when gossiping, each user picks a peer from her partial view with the *PeerSelection* function to gossip with. The type of information exchanged between peers is the membership from their partial views, selected by the *DataExchange* function. After a gossip, the *DataProcessing* funtion updates their partial views by incorporating (part) of the received membership if necessary.

Figure 1.3 illustrates the join of a new peer in a network. The peer p_6 first gossips with a peer p_3 who has already in the network. Such a peer can be discovered in various ways, including broadcasting in the local network, making use of a designated multicast group, etc. p_3 exchanges its neighbors p_1, p_4, p_5 with p_6. Then p_6 can selects p_1 and p_5 for her own partial view according to the specific application. Once p_6 becomes a member of the network, it can gossip with its neighbors to further improve its view.



(a) Gossip (b) Updated partial view

Figure 1.3: Example of gossip-based topology management

Typically, depending on the target applications, the view of each user can be managed in two main manners: (i) managing the membership at random leads to randomized topology [40, 27, 26], which ensures good properties as small distance between any two peers, small clustering coefficient, etc. [41, 26]; (ii) managing the membership based on specific criteria leads to structured

networks [28, 42, 43], which may serve a multitude of peer-to-peer applications. In our work, both the manners are considered to organize the users in collaborative tagging systems for personalizing the search.

Information dissemination In general, gossiping is a personification of efficient information propagation and gossip-based protocols have been mostly associated with the information dissemination in distributed systems.

Figure 1.4 depicts how the information is disseminated through the gossip among peers. Basically, at each cycle, the peers who possess a piece of information gossip with a peer in they know and exchange a message with it. After a few cycles, all the peers in the network can be aware of the information.

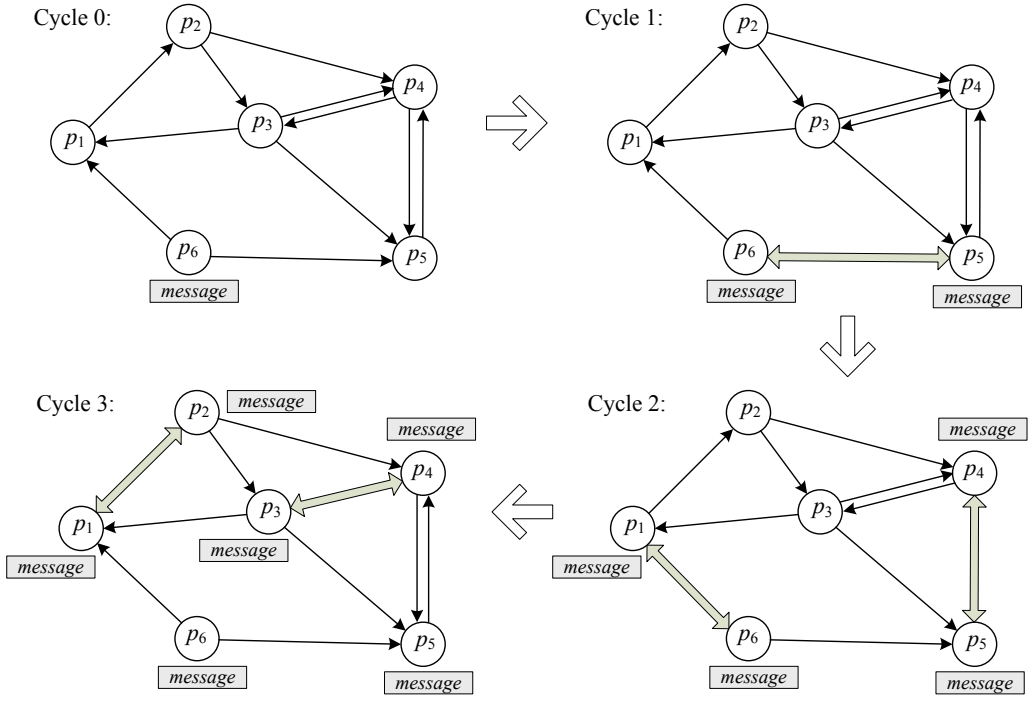


Figure 1.4: Example of gossip-based information dissemination

As shown in [44], the expected number of cycles to propagate a message to the entire network does not depend on the out-degree of the peers. The efficiency of information dissemination is further characterized in [45] in terms of replication (the number of replicas of a given information in the network at a certain moment in time) and coverage (the number of peer that have seen this information over time).

Thanks to its efficiency and scalability, we rely on gossip in the desired peer-to-peer systems to disseminate both the user-generated content and the query. Efficient search can be achieved while maximizing the utility of the information sent, received and stored by each peer in the system through gossiping.

1.2.2 Top-k processing

In collaborative tagging systems, users issue queries in the form of a set of keywords to search for the desired results. As users are mainly interested in the first few results returned by the search engines, we focus in this thesis on the top-k processing. Here we refer a result as an "object", which can be a document, a URL, a picture, a video or a user depending on the actual application. Given a query, a top-k processing algorithm aims at retrieving the k most relevant objects. The goal is typically to minimize the time it takes to come up with these objects as well as the amount of storage needed to perform the actual computation.

Among the large amount of work on top-k processing, the family of TA-style algorithms stands out as an extremely efficient method. A common premise of these algorithms is the (partial) scores of objects are pre-computed and organized in inverted lists for each term (keyword) that may appear in the query. Each entry in the inverted list of a term contains the unique score of the object for that term. Entries are sorted in descending order of their scores to enable early pruning for the low-scoring objects based on a monotonic aggregation function that determines the relevance of an object to a given query. An example of the inverted lists is given in Figure 1.5.

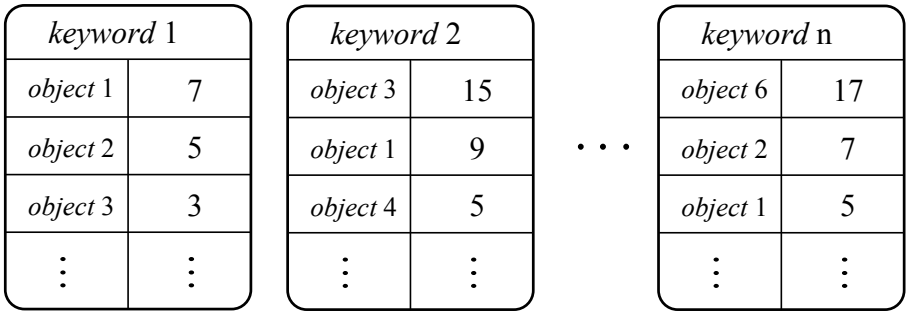


Figure 1.5: Example of inverted lists

The core idea of the TA-style algorithms is overviewed in [46, 47]. Here we only give a brief introduction of the two algorithms concerned in this thesis.

Threshold Algorithm (TA) The threshold algorithm was independently found by several research groups [48, 49, 46] around the year 2000. The main process consists of scanning the inverted lists in a breadth-first manner (sorted access) and computing dynamic thresholds during the processing to control the termination. When an object is scanned in one list under sorted access, its complete score is computed by randomly accessing its scores in other lists. A heap of k objects having the highest relevance scores is maintained. These objects are sorted in descending order of their relevance scores. Once the score of the kth object is not smaller than the threshold, the k objects in the heap form the final results. The threshold is the sum of the scores of the last seen object in each of the scanned lists. As the objects are ordered in the inverted lists, this guarantees that any objects that do not have a higher score than the threshold can not belong to the top-k results.

Figure 1.6 illustrates, through the example in Figure 1.5, how the top-2 objects for the query composed of *keyword* 1 and *keyword* 2 can be obtained with the threshold algorithm. The two inverted lists concerned by the query are scanned during the processing. When the first object (*object*

1) of the first list (*keyword* 1) is scanned, its complete relevance score is obtained by summing its scores in both lists. Then the first object of the second list (*object* 3) is scanned and added to the heap with its complete score. At the end of each step, the threshold is computed and compared to the score of the second (kth) object in the heap. As it is larger, the second object of the first list (*object* 2) is scanned. Yet, it is not added to the heap as its score is smaller than the 2 first objects. After Step 4, the threshold becomes smaller than the score of *object* 1 and we obtain the *object* 3 and *object* 1 as the top-2 results.

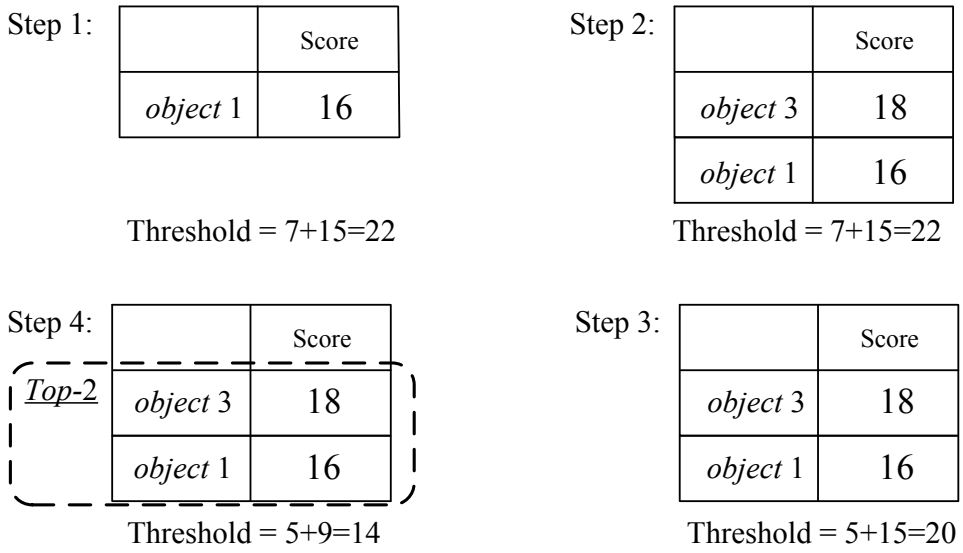


Figure 1.6: Example of TA

The threshold algorithm is optimal in a stronger sense than the previous Fagin's algorithm [50] based on inverted lists, i.e., TA is instance optimal in every database as opposed to Fagin's algorithm that is only optimal in a high-probability worst-case sense.

No Random Accesses Algorithm (NRA) There are some scenarios where the random access is not allowed or very expensive comparing to the sorted access. A number of system issues that cause random access expensive are discussed in [51]. To deal with this problem, the *No Random Accesses* algorithm [46, 52] was proposed by restricting the random access while scanning the inverted lists. In NRA, the lists are also scanned in a breadth-first manner. A best score and a worst score of each object are updated and ranked in the heap once it is accessed under sorted access. The worst score assumes that if an object has not been seen in some lists during the scanning, its scores in those lists are 0. In contrast, the best score assumes that its scores equal to the scores of the last seen object in each of those lists. The real score of an object lies between its worst score and best score. The processing stops if the worst score of the kth object in the heap is larger than the best scores of any other objects out of the first k objects. NRA is instance optimal among the algorithms that do not use random access. Figure 1.7 illustrates the top-2 processing with NRA using the same example as before.

Step 1:

	Worst score	Best score
object 1	7	22

Step 2:

	Worst score	Best score
object 3	15	22
object 1	7	22

Step 4:

	Worst score	Best score
Top-2		
object 1	16	16
object 3	15	20
object 2	5	14

Step 3:

	Worst score	Best score
object 3	15	20
object 1	7	22
object 2	5	20

Figure 1.7: Example of NRA

Both TA and NRA guarantee the correctness of the top-k objects while TA also preserves the correct order of them in the result list according to their relevance scores to the query. If we wish to know the sorted order in NRA, this can easily be determined by finding the top-1 object, the top-2 objects, etc. The advantage of TA is its bounded size of heap (k) which is independent of the size of the database. In contrast, NRA requires maintaining all the objects accessed under sorted accesses. When the heap is not a problem, NRA provides more efficient top-k processing thanks to the low cost of sorted access. We detail how these two algorithms are employed respectively to meet our specific needs in the following chapters. All the other specific techniques concerned by the proposed algorithms in this thesis will be gradually presented when the corresponding algorithm is introduced.

1.3 State-of-the-art on personalized and/or decentralized search

The contribution of this thesis lies in personalizing the top-k processing in collaborative tagging system using user-generated tags and decentralizing this personalized processing in peer-to-peer environments using gossip-based communication model. We give a high-level overview of the state-of-the-art approaches concerning the important concepts to clarify the positioning of our research in this section. As shown in Figure 1.8, we are first interested, in this section, in the existing personalization strategies. Then we introduce the search strategies in both structured and unstructured peer-to-peer networks. Finally, we overview the representative approaches that achieve personalized search in peer-to-peer networks.

1.3.1 Personalized search

Early in 2000, Lawrance [53] 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 [54] where personalized search was considered as a promising way for boosting the quality of search

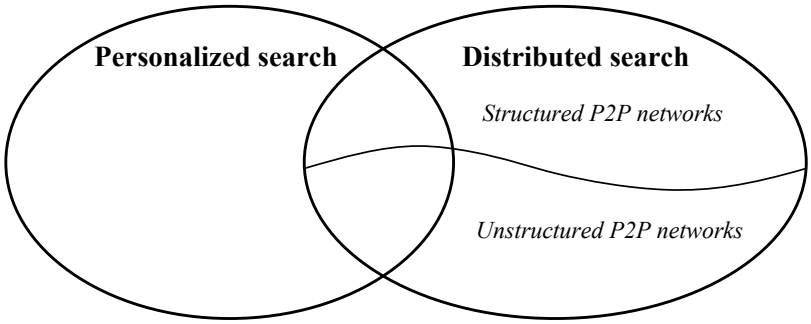


Figure 1.8: Overview of the state-of-the-art

engines. A challenge for personalization lies in learning the user's preferences that are rich enough to successfully personalize the query results. There are several prior attempts on personalizing the search.

One approach is to let users explicitly specify their interests [55, 56]. The user interests are then used to filter the search results by checking the similarity between the candidate results and their interests. For instance, in [56], users manually select several topics from the ODP hierarchy [57], which best fit her interests, to form their interest profiles. At query time, the output given by a search service (from Google, Yahoo, ODP Search, etc.) is re-ranked using a calculated distance from the user profile to each output result. Although the precision of this personalized search significantly surpassed that offered by unpersonalized search in a set of experiments, according to the studies of [58], the majority of users do not have initiatives to provide any explicit feedbacks on the search results or their interests.

As a consequence, many later works [59, 60, 61, 62, 63] attempt to automatically learn the user preferences without any user efforts to personalize the search. Typically, such preferences can be learned based on either the individual user's own activities or those of a group of users having similar interests. A wide range of user activities have been considered, including user's search histories [64, 62, 61], browsing histories [60] and tagging behaviors [63].

In [59], each user's past queries are mapped to a hierarchical tree of categories based on ODP, which represents the user's search intent. A new query can then be automatically associated to a subset of categories that serve as a context to disambiguate the words in the query. In [63], user's tagging behaviors in social bookmarking systems is used to personalize the search. In the proposed system, a user profile is a vector with n components, corresponding to a vector of tags, where the value for each tag is the number of documents tagged by the user with that tag. Each document also has a profile that is a vector of tags. The value of each tag is the number of users in the system who have tagged the document with that tag. At query time, the similarities between a user profile and the candidate document profiles are computed to rank the documents for the querier.

In most of the above personalized search approaches, only the information of the user herself is used to create her profile. There are also some approaches that incorporate the preferences of a group of users to personalize the search. Such group can be either explicitly declared [65] by joining different user groups in on-line social network services or implicitly captured [66, 67, 68, 69] by detecting the latent similar pattern in user behaviors.

The potential for using on-line social networks to enhance personalized search is investigated

though the PeerSpective prototype proposed in [65]. In PeerSpective, the Web pages visited by a dozen of researchers working in the same lab are indexed. When a query is issued, it is processed with this index. The obtained results are ranked according to their scores that reflect both their PageRank [70] scores and their scores from all the social friends who have visited them before. The user base of this experiment is very small and represents a single community of highly specialized interests. Yet, the results indicate that leveraging the interests of social friends is promising to personalize the search.

The group of users sharing similar interests can also be implicitly formed based on the past behaviors of the users. Collaborative filtering [71] is one way that data from similar users is identified for improving the search experiences for an individual user. In [60], each user's profile is derived from the browsing history of a group of users. Each user is characterized by a term-weight vector that reflects her browsing history. The similarity between two users is measured by the Pearson coefficient between their term-weight vectors. The neighbors of a user are the users who have the most similarity with her. A user's profile is computed from a weighted combination of her neighbors' term-weight vectors. Personalized search results can then be obtained by measuring their relevance to the user's profile. The evaluation in [60] shows that the user profile learned from a group of users adapts better to user preferences than the user profile based on individual user's own browsing history.

Similarly, a variety of methods for producing personalized hotlists in *delicious* are proposed in [67]. The URLs bookmarked by social friends, derived from their similar tagging behaviors, are used to generate the hotlists. These social friends are obtained based on the overlap on the tagged URLs, the overlap on the used tags or the overlap on both the URLs and the tags. Their evaluation shows that users' tagging behaviors can be used to derive implicit social ties and such ties are good indicators of user preferences, which ensures higher relevance than global processing.

A large-scale evaluation and analysis of personalized search strategies are conducted in [72]. The profile of each user is a weighting vector of some pre-defined topic categories built with her click history. Each page is also presented as a weighting vector of the same pre-defined topic categories. Personalized search results are obtained by either person-level ranking or group-level ranking. In the personal-level ranking, the pages are ranked according to their cosine similarity to the user's profile. In the group-level ranking, the relevance of a result is computed by merging the clicks of each group user according to her similarity with the user who issues the query.

These group-based personalization approaches differ from each other mainly in the ways they measure the similarity between users and they score the relevance of each candidate result. Differently, a general indexing and query processing framework which applies to a wide class of similarity measures and scoring functions is developed in [12]. This work focuses on the trade-off between storage and processing time to provide personalized top-k processing. Since it serves as a baseline of the algorithms proposed in our work, we will detail this approach in the next chapter.

Table 1.1 summarizes the characteristics of the above approaches according to the way the personalization is carried out. Basically, the personalization based on a group of users provides better results than that solely relies on the querier's interests. The previous works also show that the implicit relationship between users is very effective to be leveraged for improving the result quality. Although the *Explicit* and the *Implicit* approaches were not directly compared, we believe that we can learn a lot from people we may not know but with whom we share many interests. Therefore, our research focuses on the implicit relationship (*Implicit*) with a group of users having similar interests (*Group*) for personalizing the search.

Table 1.1: Characterization of the state-of-the-art personalized search approaches

	Explicit[1]	Implicit[2]
Individual[3]	[56], [55]	[59], [62], [61], [63], [72], [64]
Group[4]	[65]	[60], [66], [67], [68], [69], [12], [72], [62]

[1] user preferences are explicitly declared by the user.
[2] user preferences are implicitly derived from their past behaviors.
[3] user preferences are based on her own past behaviors.
[4] user preferences are based on a group of users having similar preferences.

1.3.2 Distributed search

With the increasing amount of content sharing applications in peer-to-peer systems, an emerging challenge is to support efficient search in such highly distributed environments. Previous works focus on top-k processing in both structured and unstructured peer-to-peer networks.

Key techniques in distributed top-k processing In peer-to-peer networks, data are distributed among peers and each peer is only in charge of a subset of data. Several algorithms have been proposed to process the top-k queries in such environments. A common assumption underlying these approaches is that the data are vertically distributed over peers, where each peer provides a ranking of the data they have over some attributes.

The algorithm TPUT (Three-Phase Uniform Threshold), proposed in [73], aims to prune unnecessary data objects transmitted between the peer processing the query and those possessing the data. This is achieved through a three phase algorithm. In the first phase, each peer forwards to the peer which processes the query the top-k objects in its local index according to the query attributes it has. This allows to compute a relevance score lower bound for the final top-k objects. In the second phase, the objects having a relevance score larger than the assigned threshold for each attribute are sent to the peer who processes the query. In the last phase, this peer computes the relevance scores of all the objects it has received to obtain the final top-k objects. TPUT reduces network traffic by one to two orders of magnitude compared to applying the existing TA algorithm in distributed networks. TPAT [74] is a modification of TPUT where the threshold for each attribute is adapted to the distribution of the relevance scores in its inverted list. Yet, as stated by the authors, this approach may incur very high computational cost.

KLEE [75] improves TPUT through a combination of histograms and Bloom filters to reduce the communication costs. When a peer is required to return its local top-k objects, it piggybacks a histogram of the local relevance score distribution and the Bloom filters that summarizes the objects in each of the histogram intervals. The peer who processes the query can then combine this information to derive a higher threshold than TPUT would have. The performance of algorithms in the TPUT family are further optimized in [76]. In this approach, peers are organized in hierarchical groups in the second phase of the processing according to the information they send to the peer which processes the query. Each intermediate peer in the resulted hierarchy only aggregates the objects from its children, which reduces the bandwidth consumption. An adaptive mechanism is also used to choose different threshold for each peer.

The algorithms of TPUT family have been designed for general networks without any assumption on network topology. Once the right peers for processing a query are identified, efficient top-k

processing can be carried out in either structured or unstructured peer-to-peer networks.

Search in structured peer-to-peer networks In structured peer-to-peer networks, the peers to process a query, e.g. those owing the query related attributes, can be achieved using Distributed Hash Tables (DHT).

A distributed hash table allocates the association $\langle key, value \rangle$ among the peers in a network. Each peer is in charge of a set of keys. According to the applications, the key and the value may represent different things. For instance, in a file sharing system, a key is the identifier of a file and its value is a list of peers who possess this file. In a system that provides top-k processing, a key is an attribute that characterizes the objects and its value is a list of objects related to this attribute. Distributed systems based on DHTs are very efficient and scalable to support keyword search. The peers in the systems form structured networks where their neighbors are selected according to the keys they are in charge of. The search in DHT corresponds to first find the peers who are in charge of a given key and then query its values owned by each of these peers.

CAN [77], Chord [78] and Pastry [79] are the most representative systems that rely on different forms of DHT to enable efficient search. CAN [77] is based on a d-dimensional Cartesian coordinate space in a d-torus. The entire coordinate space is partitioned among all the peers in the system so that each peer owns its own distinct zone within the whole space. Each $\langle key, value \rangle$ pair is mapped to the coordinate space and stored by the peer who owns the corresponding zone within in which the pair lies. The users whose zones are adjacent to a user serves as its routing index. The query is forwarded in CAN in a greedy manner: each peer who receives the query sends it to an adjacent peer whose zone contains the $\langle key, value \rangle$ whose key is the most similar to the query. The complexity of routing in CAN is $O(d \times N^{1/d})$, where N is the number of peers in the network. Chord [78] is based on a logical ring constituting the peer identifiers from 0 to $2^b - 1$. b should be large enough to avoid collisions. Each $\langle key, value \rangle$ pair is assigned to the first peer in the clockwise order of the ring whose identifier is equal or larger than that of the key. This peer is called the successor of that key. The routing table of a peer whose identifier is n consists of at most b ($O(logN)$) other peers. Each peer is the successor of the key whose identifier is $n + 2^{i+1}$ ($1 \leq i \leq b$). With greedy routing, the number of peers that must be contacted to find a successor in a network of N peers is $O(logN)$. Pastry [79] is also based on a circular space where each peer is randomly assigned a 128-bit identifier ranging from 0 to $2^{128} - 1$. Each peer is responsible for the set of $\langle key, value \rangle$ pairs whose identifiers are the closest to its own. Pastry uses a prefix-based routing. Messages are routed to the peer whose identifier is numerically closest to the given key, i.e., having the most common prefix as the searched key, in less than $O(log_{2^b} N)$ steps. b is a small parameter indicating the number of bits solved at each step.

DHTs are efficient for searching in the sense that any key can be exhaustively retrieved within $O(logN)$ steps. However, the most widely used DHTs only provide answers as "found" or "not found" according to the exact match. More recently, the data management community has focused on the extending such architecture based on DHTs to support more complex queries, i.e., the top-k queries [80, 81, 82, 83].

In Minerva [82], a Chord-based DHT is used to partition the term space. Each peer publishes per-term summaries of its local index to the directory. The DHT determines the peer currently responsible for each term summary. Each peer maintains for each term a PeerList that contains all the peers who have posted a summary for that term as well as the necessary statistics for the query processing. At query time, the query is first routed to the peers responsible of the query terms to

obtain a set of PeerLists. The most promising peers for processing a query is computed from these PeerLists. Finally the query is processed with the local index of these peers. Any algorithm of the TPUT family [73, 74, 75, 76] can be used in this context.

In pSearch [80], peers are organized using CAN. Each peer is responsible for storing the index containing certain terms. Given a document, a term vector is computed using latent semantic indexing (LSI) [84]. In this way, documents that are semantically close to each other are stored logically close to each other in CAN. Query is routed to the proper regions, much smaller than the case where the documents are randomly distributed, to obtain the top-k results.

Search in unstructured peer-to-peer networks Without imposing any stringent constraints over the network topology, unstructured peer-to-peer networks can be constructed more efficiently comparing to structured peer-to-peer networks. Yet, the lack of structure leads the efficient search more difficult.

In purely unstructured peer-to-peer networks like Gnutella [85], blind search through flooding is usually used for content discovery. To find a file, a peer sends the query to its neighbors and these neighbors continue to forward the query to their neighbors until the query reaches some pre-defined radius. Despite its simplicity, flooding is in general not very scale, since in a large network the successful search may decrease dramatically without significantly enlarging the flooding extent.

To improve the search performance, some guided search approaches have been proposed [86, 87]. Basically, queries are not necessarily flooded through the entire network anymore, but can be purposefully routed to the relevant parts of the network. In [86], a variant of flooding is proposed. The query is only forwarded to the peers who have not received it before. Each peer processes the query with its local index, merges the sorted result lists from its children and sends back its result list to its parent. The querier only needs to merge the result lists from its direct neighbors to obtain the final top-k results. PlanetP [87] uses gossip to globally replicate a membership directory. Each peer maintains an inverted index of its documents and spreads the term-to-peer index. Based on this replicated global index, a searching peer first identifies the set of peers having the query related terms with the global index and then ranks the relevant documents returned by these peers to determine the most pertinent ones with a TFxIDF-ranking algorithm.

Another possibility is to exploit interest-based locality as a search guidance [88, 89, 90, 91, 92, 93]. The basic assumption of these approaches is that if a peer has a file required by another peer, it is also likely to have other files required by the same peer. For instance, in [91], the similarity between two peers are derived from the fraction of documents they have for each topic. Each peer maintains both short-links and long-links to according to their similarities, constituting a small-work network. The query is then routed in a greedy manner following these links to achieve efficient top-k processing. In [88], shortcuts among peers are established according to previous successful queries.

Some works [94, 95, 96] rely on super-peers to forward and process the top-k queries. The approach proposed in [94] aims to reduce the number of objects transmitted among super-peers. Each super-peer manages an index containing the information about the contribution of its local peers and adjacent super-peers for answering recently posed queries. Query is first routed in a spanning tree rooted at the querying super-peer to the super-peers who have recently answered the same query. Each super-peer then processes the query in its inner network and sends the possible results back to the querying super-peer. Similarly, in BRANCA [95], network paths are pruned according to result caching. The core of BRANCA is the integration of a semantic caching of query results and a routing index. Specifically, the semantic caching improves conventional

cache-replacement schemes by retaining data that are more likely to be retrieved by subsequent queries. The routing index tags additional information to each entry in a routing table, which serves as a brief summary of the data in the subnet corresponding to that entry. In BRANCA, each super peer caches data fetched from various subnets in the past top-k search. Given a new query, the cache allows a super-peer to forward the query only to a small part of the network that may contain the final results. SPEERTO [96] also relies on super-peers and explores a skyline-based routing. The top-k queries are forwarded among super-peers using skyline operators.

These search approaches in peer-to-peer networks aim to provide efficient top-k processing with low network traffic. In structured peer-to-peer networks, the documents are indexed in a global manner and the indexes are distributed among peers. The querying peer can identify the peers responsible for the queried terms in very limited steps. Yet, the management of the global index requires substantial bandwidth consumption and the stability of the peers in the system. In contrast, the search in unstructured peer-to-peer networks is more flexible. In such networks, each peer locally manages the documents it publishes. Well designed indexing and routing techniques ensure efficient search with low network traffic. We thus design our algorithms in this thesis on top of unstructured peer-to-peer networks. The previous approaches in unstructured peer-to-peer networks are efficient in terms of response time and bandwidth consumption. None of them is however personalized. Actually, personalization is critical in collaborative tagging systems for disambiguating the queries and fulfilling the individual needs. For this reason, we try to perform personalized top-k processing in unstructured peer-to-peer networks.

1.3.3 Personalized search in distributed systems

In fact, there are indeed a few works [97, 98, 99, 100] in distributed systems that consider personalization for improving the search accuracy.

In [97], personalized search is achieved in a peer-to-peer network through personalized PageRank [101]. The personalized PageRank is computed in a distributed manner and each peer is aware of the ranks of the pages owned by its neighbors. Upon issuing a query, the peer first forwards the query to its neighbor having the highest ranked page according to its personalized PageRank. Each peer receiving the query continues to forward the query to its unvisited neighbor with highest ranked page until the desired number of results is obtained.

MAAY [98] is a decentralized personalized search engine that aims to provide personalized results by taking into account the common interests between peers and the thematic similarity between documents. The peers, documents and words are organized in a bipartite graph, where links exist between documents sharing common users and documents sharing common words respectively. MAAY peers learn locally from their previous interactions the profiles of others peers and use feedbacks to raise words characterizing documents. Using these profiles, the querying peer can choose peers to query and rank documents according to its own profile. The similarity between a peer's query and a document is measured by the combination of the distance between this document and all the other documents owned by this user containing the query words in the graph derived from the queried peers. This graph-based approach provides good approximation of personalized results to its users in a peer-to-peer way.

In [99], different routing strategies are proposed to personalize the top-k processing based on the peer-to-peer Web search engine Minerva [82]. The key idea is for each querying peer to select a few

most promising users in the system to process their top-k query. Both the social query routing and the spiritual query routing lead to personalized results. The social query routing simply relies on a set of explicit friends to process the query. The spiritual query routing selects target peers based on the behavioral affinity. An information-theoretic measure, Kullback-Leibler divergence [102], on the tag frequency distribution of a peer's bookmarked pages is used to quantify this affinity. The Minerva framework is extended by keeping at each peer a list of such peers to enable either social or spiritual routing. Yet, it is unclear that how the behavioral affinity can be captured using DHT in Minerva.

The common goal of these approaches in peer-to-peer systems is to take advantage of the social aspects to personalize the search. The approach proposed in [97] focuses on the Web search. The employed personalized PageRank is not applicable in collaborative tagging systems, where the links among items do not exist. In MAAY, peers rely on their previous requesting and responding interactions to identify the similar peers for query processing. The personalization highly depends on the activity level of each user. Inactive users that rarely send queries can hardly find good users to personalize their search and this is undesirable. In [99], several strategies that personalize the top-k processing through social routing are compared. But it focuses more on the result quality under each strategy. It is unclear that how the behavioral affinities among users can be actually captured. Our work extends this line of research that personalizes the search in peer-to-peer networks. We mainly focus on how to identify and leverage the social aspects in top-k processing so that the result quality is independent of the querier's activity level. More importantly, different from the previous work, we specifically design algorithms that are adequate for collaborative tagging systems.

1.4 Conclusion

In this chapter, we clarified the context of our research and reviewed some existing works that improve search quality through personalization. In general, the search results are personalized according to either the querier's own preferences or those of a group of promising users. Such user preferences can be explicitly declared by each user or implicitly captured through their past behaviors. We focus in this thesis on the implicit affinity among a group of users as it appears to be the most promising way to provide high quality personalization. Yet, the previous works mainly rely on the past user behaviors to deduce their preferences, which makes it difficult to fit the emerging interests of the queriers. We thus intend to propose a new scheme of personalization that is appropriate for all kinds of queries.

Our main purpose is to provide personalized query processing in fully decentralized environments. This is because the scalability problem faced by centralized systems may be significantly alleviated. In peer-to-peer systems, information is distributed among peers, so that each peer only has to maintain a very limited amount of information and more queries can be tackled at the same time. This also avoids the central authorities abusing the personalized information at their disposal. In peer-to-peer systems, more efforts are dedicated to improve the search efficiency. We carry out our research in unstructured peer-to-peer systems since they do not rely on any global index and are very flexible in the face of peer dynamics. However, to the best of our knowledge, only a few works address the personalization issue. Unfortunately, these approaches are not satisfactory enough as they either impose constraints on the user activities or require additional links among data items. That is why we devote to the algorithms that achieve personalized query processing in peer-to-peer systems in our work.

CHAPTER 2

OFF-LINE PERSONALIZED QUERY PROCESSING

2.1 Introduction

Collaborative tagging systems represent huge mines of information but make their exploration challenging because of the unstructured nature of the tagging and the lack of fixed ontology. An appealing way to disambiguate the exploration process 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*. In contrast, a *Keanu Reeves* fan may just look for this movie. Personalization based on user affinities has the potential to disambiguate these situations.

Several personalized approaches have been proposed to leverage social networks in search procedures [99, 65]. 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 people you might not know, but with whom you share many interests, can be very helpful when searching the web.

Very elegant approaches have recently been proposed to capture and leverage this implicit personalization through social scoring models that capture the user-centric correlation among different tags in order to improve information retrieval quality [66,103]. But getting these to work is challenging. First, maintaining the fine-grained information on a user basis is extremely space consuming. Under the estimation of [12], several terabytes are necessary to personalize the top-k processing in a *delicious* system of only $100,000$ users. Considering that the actual system has millions of users, a centralized personalization approach seems cumbersome. Moreover, the users in collaborative tagging systems are very active and they continue changing their profiles by tagging new content. This makes it difficult for a centralized system to capture these changes timely.

As a consequence, leveraging implicit social networks in the search process calls for decentralized solutions. Besides being scalable and able to cope with dynamics, decentralized solutions circum-

vent 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. Therefore, we aim to provide efficient and scalable personalized query processing, based on implicit social networks, in a peer-to-peer manner.

This chapter is organized as follows. Section 2.2 describes the system model and highlights our motivation of performing the off-line personalization in peer-to-peer systems. Section 2.3 introduces the algorithm P3K, our first step toward a fully decentralized and personalized query processing scheme. Section 2.4 presents an advanced algorithm, namely P4Q, to address the storage, bandwidth and dynamics issues in the decentralized environments. Section 2.5 concludes this chapter.

2.2 Social network model and background

2.2.1 Social network model

We consider collaborative tagging sites to be based on as an information space $\mathbb{U} \times \mathbb{I} \times \mathbb{T}$, where $\mathbb{U}$ denotes the set of users, $\mathbb{I}$ contains the items in the system and $\mathbb{T}$ is the set of all related tags. $Tagged(u, i, t)$ captures the fact that a user u tags the item i with the tag t. Each user has a profile that conveys her endorsement of visited items by tagging them. The profile of a user u is described as a set of her tagging actions, i.e.,

$$Profile(u) = \{\langle i, t\rangle | \, Tagged(u, i, t)\}.$$

The whole network is modeled as a directed graph where each node (peer) corresponds to a user and an edge represents a link between two users. When there is a directed edge from user u_i to user u_j, u_j is considered as a neighbor of u_i. All the neighbors of u_i, which can be either explicitly declared like friends in *Facebook* or implicitly captured through similar tagging behaviors, form her social network. In this work, we are only interested in the implicit relationship between users, i.e., the implicit social networks.

We refer such implicit social network as *personal network*, noted as $Network(u)$, in the following to not confuse with the traditional sense of social network as neighbors in our personal network are those with similar tagging behaviors and might be and remain unknown to the actual user. Figure 2.1 depicts our social network model. As we can see, the user u_2 is considered as a neighbor in the user u_1's personal network because they have similar profiles.

2.2.2 Off-line personalization

We consider a query $Q = \{t_1, ..., t_n\}$, issued by a user u_i with a set of tags $t_1, ..., t_n$. The personalized query processing for Q aims to return a set of items having the highest relevance scores from u_i's personal network. This relevance score is thus user-specific and network-aware as shown in Figure 2.2(b). In contrast, in a traditional non personalized approach, all the users in the system would be considered for the query processing (Figure 2.2(a)). The personalization lies in limiting the search space within a subset of the system, namely the relevant users to the querier.

We refer this as an off-line personalization since the implicit social network of each user is built independently of the query processing. In other words, the social network of each user is precomputed and known at her query time. Once a user issues a query, the system only needs to process this query within the user's social network to obtain the personalized query results.

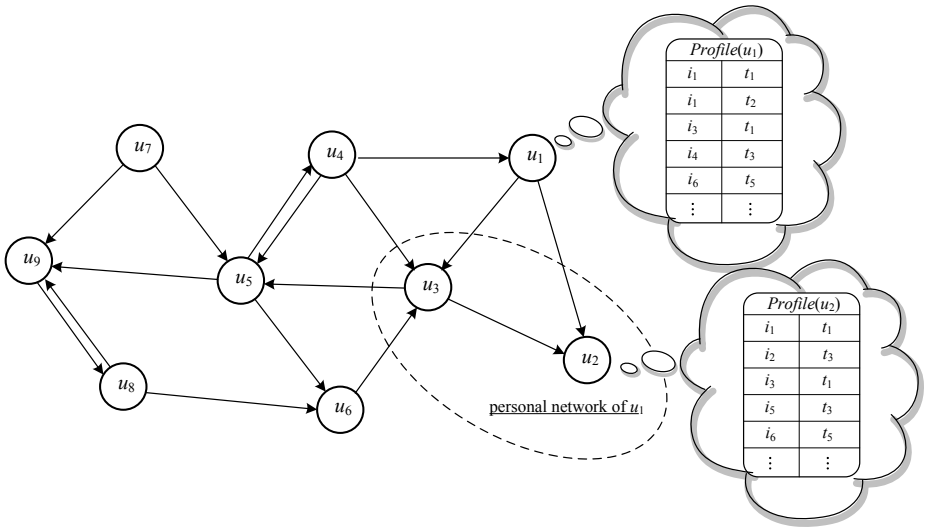
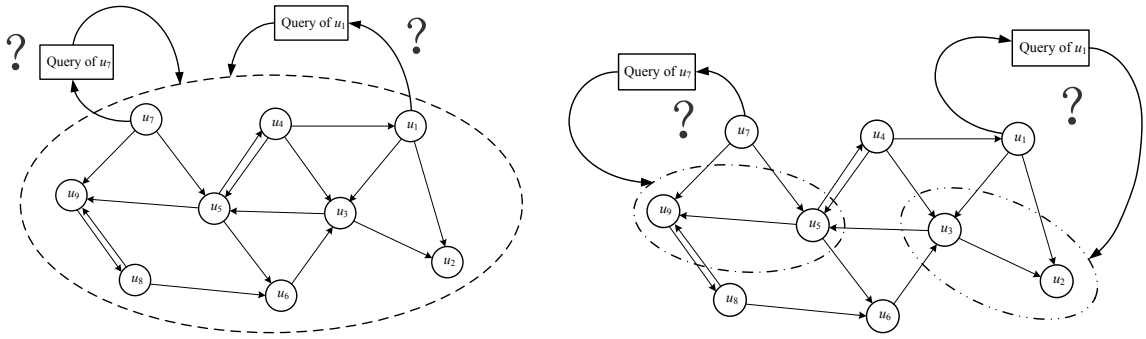


Figure 2.1: Social network model

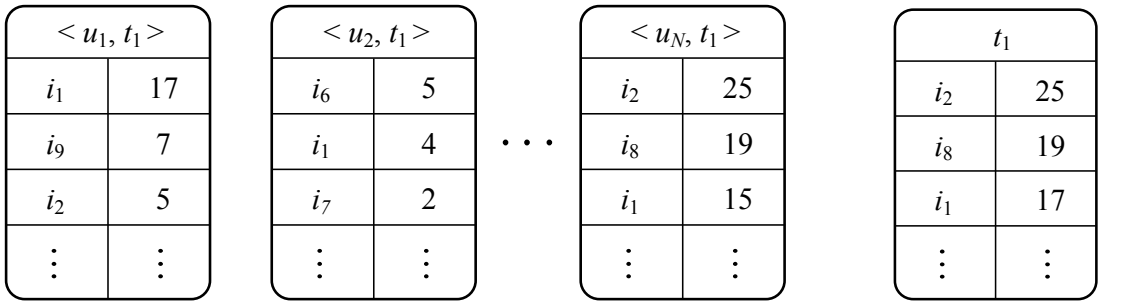


(a) Non-personalization

(b) Off-line personalization

Figure 2.2: Rationale of personalization

Such off-line personalized query processing was first proposed by Amer-Yahia et al. [12] for centralized collaborative tagging systems. The most straightforward strategy in their work, called *Exact*, is to build one inverted list for each $\langle tag, user \rangle$ pair. Query Q generated by a user u_i is processed on the $\langle t_n, u_i \rangle$ lists, where $t_n \in Q$, using a traditional top-k processing algorithm such as NRA and TA. However, storing these lists is prohibitive space-wise for a single server. As shown in Figure 2.3(a), if the tag t_1 is used in all the users' personal networks, as many as inverted lists should be maintained for t_1, which highly duplicates the information to store. Therefore, they explore another strategy, *Global Upper-Bound*, which maintains user-independent inverted lists. In the inverted list of a tag, only the maximum score over all personal networks for each item is maintained. The exact score of an item for a user's query within her personal network is computed at query time. With *Global Upper-Bound*, a considerable storage space is saved as only one list is maintained for each tag (Figure 2.3(b)) but much more time is required for query processing. To strike a balance between the two extremes, users are then clustered according to their similarities



(a) Exact (b) Global Upper-Bound

Figure 2.3: Centralized off-line personalization

with each other. Only score upper-bounds over all clusters are maintained in per $\langle tag, cluster \rangle$ pair inverted list. However, the proposed strategies have to trade the processing efficiency for the storage capability. The efficiency highly relies on the right clustering and is not very flexible with a dynamic set of users.

In this chapter, we take these challenges, faced by these centralized solutions in terms of storage and processing time, up in a fully decentralized and gossip-based way to achieve the off-line personalized query processing. We use the term "user" to mean a user in the collaborative tagging system and its associated underlying machine in the peer-to-peer system and generally refer to the canonical user as "she".

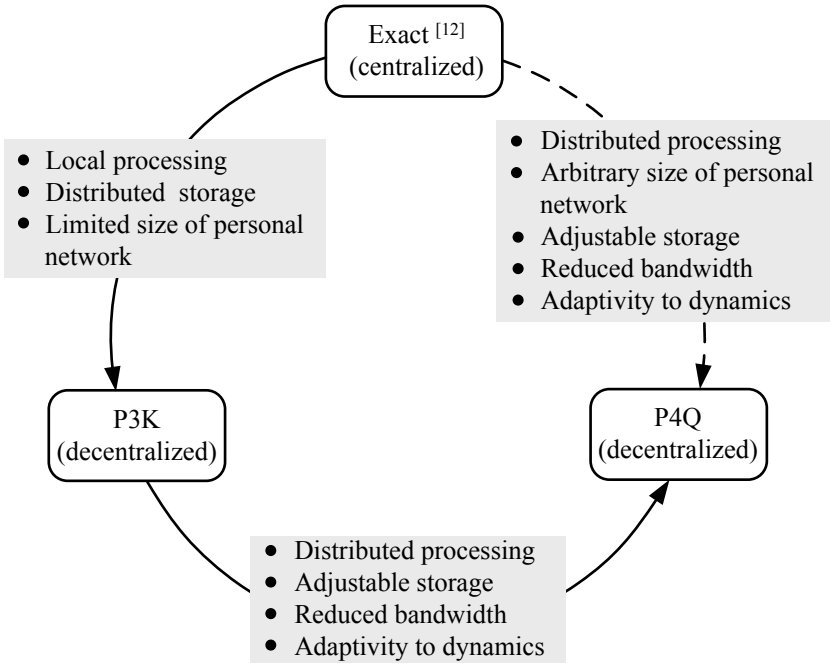


Figure 2.4: Organization of the work on off-line personalized query processing

As shown in Figure 2.4, we first introduce the algorithm P3K, which decentralizes the *Exact* strategy proposed in [12]. More specifically, the personal network and the associated per $\langle user, tag \rangle$ inverted lists are stored by each user. The query is locally processed within each user's personal network. The size of personal network is further limited to ensure the scalability of the system and it approximates well the result quality that can be obtained with *Exact*. In order to render users the freedom to adjust the storage they offer for the query processing and optimize the bandwidth consumption of the system, another algorithm P4Q is further proposed. In this algorithm, the top-k queries are gossiped and collaboratively processed in a distributed manner. In addition, P4Q contains a mechanism that proactively reacts to the dynamics involved in the system in terms of interest evolving and user departing. We detail these two algorithms in the following sections.

2.3 Toward personalized peer-to-peer query processing

In this section, we present P3K (Personalized Peer-to-Peer top-K processing), our first step toward a decentralized and personalized query processing protocol that accounts for implicit user affinities. The objective of P3K is to assess the feasibility of the off-line personalized top-k processing in a peer-to-peer environment.

In P3K, each user maintains a set of neighbors that form her personal network. A query is processed with the information available in the querier's personal network to get personalized query results. This full control of the personal network allows for a full personalization of the query processing. More importantly, the processing and storage capability increases linearly with the number of users avoiding the potential bottleneck at a single server.

There are many possibilities of establishing the neighborhood relationship between two users according to their implicit affinities. We do not assume any particular metric to define the similarity between two users. Various notions of similarity [67] can be exploited according to the specific applications. Here, we use, for the sake of comparison, the criterion in [12] that a user u_j is a neighbor of the user u_i if and only if u_j tagged a sufficient number of items with at least one same tags as u_i, i.e.,

$$|\{i|\exists t \in \mathbb{T}, Tagged(u_i, i, t) \wedge Tagged(u_j, i, t)\}| > threshold. \tag{2.1}$$

In this setting, a key problem is how to efficiently discover each user's personal network in a peer-to-peer way, i.e., converging to the right set of neighbors as quickly as possible to guarantee the quality of top-k results.

2.3.1 Personal network construction

At the heart of P3K lies the sub-protocol that provides each user with a personal network. For the sake of presentation simplicity, we first assume no restriction on the size of the personal network of a user. Then we will discuss how we limit this size and maintain only a relevant fraction of this.

2.3.1.1 Unlimited personal network

An unlimited personal network contains all the users sharing similar tagging behaviors as neighbors according to the metric described in Formula 2.1. Namely, two users are considered neighbors if the number of items they tag using the same tags reaches a given threshold. This threshold can be

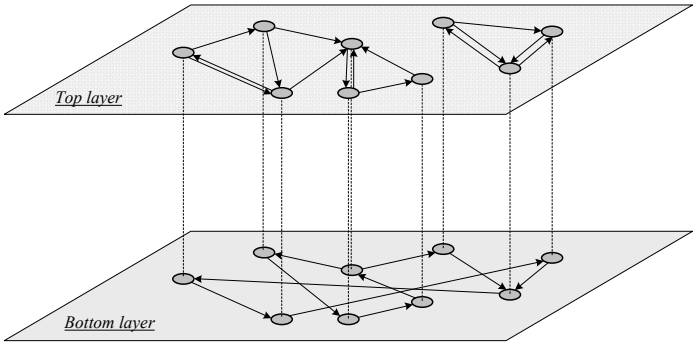


Figure 2.5: Two-layer network model

trained for each user according to the equilibration between its value and the corresponding result quality of previous queries.

The personal network of each user is discovered and maintained through a two-layer gossip protocol (Figure 2.5). The top layer is in charge of tracking the similarity between users and discovering new neighbors. All the neighbors of a user u in the top layer form her *personal network*. Underlying the top layer, a bottom layer, typically known as a random peer sampling (RPS) [26], provides each user with r neighbors picked uniformly at random from the system. This second set of users is called the *random view*. As different user interests may lead to disjoint personal networks, the random view is maintained to keep the whole network connected. Once the top layer has stabilized, the bottom layer enables the users to discover new neighbors through RPS in case of changes in a user profile.

The bottom layer and the top layer run in parallel, i.e., at the beginning of each cycle, a user gossips with both a neighbor from her random view and a user from her personal network. The pseudo-code of the gossip protocol is presented in Algorithm 2.1. We use $view(u)$ to stand for either the personal network or the random view of a user u for the ease of presentation. u_{init} is the user who initiates the gossip and u_{dest} is the user gossiping with u_{init}. Table 2.1 compares in detail how the three functions, described in Section 1.2.1 that characterize a gossip, are implemented in both layers for maintaining the personal networks.

Table 2.1: Characterization of the two-layer gossip protocol

Function	Bottom layer	Top layer
PeerSelection()	select neighbor with oldest timestamp from random view	select neighbor with oldest timestamp from personal network
DataExchange()	send a subset of random neighbors from random view	send a subset of random neighbors from personal network
DataProcessing()	keep r random users for random view	keep all users matching Formula 2.1 for personal network

Note that a timestamp is used for each neighbor and the neighbor with oldest timestamp is always selected as gossip destination. Once a user is picked, its timestamp is set to zero and other neighbors' timestamps are increased by 1. The timestamp ensures that all the neighbors have a comparable chance of participating in gossiping. Neighbors in personal networks are exchanged

Algorithm 2.1 Personal network construction through gossip

Active Thread
for each cycle **do**
  u_{dest} = PeerSelection($view(u_{init})$)
  $dataToSend$ = DataExchange($view(u_{init})$)
  Send $dataToSend$ **to** u_{dest}
  Receive $u_{dest}.dataToSend$ **from** u_{dest}
  $view(u_{init})$ = DataProcessing($u_{dest}.dataToSend$, $view(u_{init})$)
end for

Passive Thread
loop
  Receive $u_{init}.dataToSend$ **from** u_{init}
  $dataToSend$ = DataExchange($view(u_{dest})$)
  Send $dataToSend$ **to** u_{init}
  $view(u_{dest})$ = DataProcessing($u_{init}.dataToSend$, $view(u_{dest})$)
end loop

between gossiping users in the top layer. The underlying intuition is that neighbors' neighbors are likely to be neighbors as they all share similar interests. This enables the users to find their personal networks quickly as we will see in Section 2.3.3.2.

2.3.1.2 Limiting the size of personal network

Obviously, an unlimited personal network may grow to infinity as the number of users involved in the system increases. This may generate a lot of information exchanges and increase storage requirements. In practice, a carefully selected subset of users should be representative enough to provide most of the useful information for top-k processing. We hence bound the size of the personal network by a number s so that only the s users with the highest similarity are involved in the personal network of a given user.

In collaborative tagging systems, the similarity between users depends on the tagging behaviors exhibited in their profiles. The overlap among the tags in different user profiles reflects their common interests on topics. Instead, the overlap among tagged items reveals their similar preferences on specific items. Because a tag can be used for several items, and the same item can be tagged differently by different users [104], we compare tagging behaviors by comparing the number of $\langle item, tag\rangle$ pairs in common rather than the number of items tagged by both users with the same tags. The intuition is that the more common tags are used for an item, the more similarly the users understand and judge the world. We thus refine the similarity between two users u_i and u_j as

$$Similarity(u_i, u_j) = |\{\langle i, t\rangle | Tagged(u_i, i, t) \wedge Tagged(u_j, i, t)\}|. \tag{2.2}$$

For each candidate neighbor u_j of user u_i, we re-compute their similarity and keep the s users having highest similarity with u_i as neighbors. The personal network formed with such refined similarity constitutes the P3K personal network.

2.3.2 Inverted lists and query processing

We use the same scoring function as in [12] to rank the items for the sake of comparison. The score of an item i for user u_i and tag t_n is defined as the number of users in u_i's personal network who tagged i with t_n, i.e.,

$$Score_{u_i,t_n}(i) = |\{\langle u_j, i, t_n\rangle | \forall u_j \in Network(u_i), Tagged(u_j, i, t_n)\}|.$$

The overall score of an item i for user u_i's query Q, composed of a set of tags t_1, ..., t_n, is the sum of all the scores related to the query, i.e.,

$$Score(Q, i) = \sum_{t_n \in Q} Score_{u_i,t_n}(i).$$

Alternative functions can be used to compute such a user-specific score.

In the process of gossiping, the profiles of a querier u_i's neighbors are stored by herself. To enable efficient top-k processing, inverted lists for each $\langle user, tag\rangle$ pair are also computed and stored by the user. The inverted list of the pair $\langle u_i, t_n\rangle$ contains all the items that are tagged by at least one neighbors in u_i's personal network with the tag t_n. The score of each item i in that list is $Score_{u_i,t_n}(i)$. The items are sorted in descending order of their scores to form the inverted list of $\langle u_i, t_n\rangle$.

The inverted lists are constructed lazily so that an inverted list is computed only when it is necessary for the query processing. There is no need to pre-compute the inverted lists for the tags that may never be queried as in a centralized case, especially before the personal networks stabilize. When new neighbors are found through gossiping, the information in the profiles of users who no longer belong to the personal network is removed from the corresponding inverted lists if they have been added before.

When a user generates a query, she first checks whether the inverted lists for the tags in the query already exist and determines whether they should be updated to reflect new neighbors. If new neighbors do exist, the information in their profiles is added to the inverted lists. Once all the related lists are up-to-date, the user processes her query locally with NRA (Section 1.2.2) to get the top-k results.

2.3.3 Experimental evaluation

In this section, we report on the evaluation of P3K and compare it against the state of the art centralized version [12]. We first describe in Section 2.3.3.1 the *delicious* dataset used for the evaluation and the evaluation metrics. In Section 2.3.3.2, we assess P3K qualitatively, i.e., the relevance of the personal networks through the quality of the top-k processing results. We compare P3K with the state of the art centralized personalized top-k approach [12]. We then proceed to the quantitative evaluation (Section 2.3.3.3) of P3K with respect to space requirement, response time and scalability.

2.3.3.1 Experimental setup

Data set and query generation The evaluation of P3K has been conducted in *PeerSim* [105], 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 behaviors follows a long tail distribution as most items and tags are used by few users [106]. 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[1]. 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.

Evaluation metrics As a baseline for comparison we use the two extreme approaches presented in [12]: one of them, *Exact*, does not limit the storage space but exhibits a low processing time; an alternative one, *Global Upper-Bound*, requires significant processing time but reduces storage space. Our goal is to show that P3K achieves the best of both approaches.

We first evaluate the ability of P3K to discover users sharing similar tagging behaviors with respect to quality, convergence and impact on the top-k results. 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 service. The personal network is then gradually built and maintained through the gossip protocol.

The personal networks are compared to ideal ones[2] obtained off-line using the global information about all users' profiles. We measure the success ratio as the number of neighbors that are in the personal network (and should be) over the total number of neighbors in the ideal personal network. Before evaluating the impact of the quality of the personal network on the top-k results, we measure the number of cycles required to converge to a stable personal network with unlimited size.

We evaluate the speed of convergence with the average of the resulting success ratios over all users for each cycle, i.e.,

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

$success\_ratio$ reaches 1 when all users find their ideal personal networks. This metric cannot be applied to the approach described in [12] where the personal network is given to each user by the system (namely pre-computed and stored in the central database).

To evaluate the top-k results, we compare P3K with the results of Exact ([12]) as the baseline. Note that all algorithms proposed in [12] provide the same top-k items and only differ in processing time and storage space.

[1] 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 expressions, like *socialnetwork*, *socialsearch*, *socialresponsiblility* etc., which give a more precise description of the items. This also gives a hint of the ability of the tagging vocabulary to grow infinitely.

[2] By ideal, we mean the same as the one obtained in a centralized system using the entire set of profiles (*Exact* scheme in [12].)

We use *recall* [107] to evaluate the quality of top-k results. The recall R_k is the proportion of the total number of relevant items that are retrieved in the top k:

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

Recall quantifies the coverage of the result set and varies between 0 and 1. In this context, an ideal $R_k = 1$ means that P3K achieves the same top-k results as *Exact*.

Each user stores the profiles of her neighbors and builds inverted lists accordingly. The space requirement for each user is measured as the number of entries in the inverted lists and the number of entries in the profiles of her neighbors. The response time of a query is dominated by the query execution time over related inverted lists because queries are processed locally and there is no delay due to network transmission. The execution time for a query is quantified by the number of sequential accesses of related inverted lists, which is considered as an objective metric in [12].

2.3.3.2 Qualitative P3K evaluation

Unlimited personal network evaluation We assume here that the size of personal networks is unlimited. For the sake of comparison, we use the same similarity metric between users as in [12]: two users are neighbors if they have at least 2 items tagged with the same tag. Any user matching the criterion is integrated in the personal network of a user. We evaluate the convergence properties of the personal networks by measuring the number of gossip cycles required for users to build their personal networks.

Two *gossipSize*, 20 and 50, are considered (the bottom layer and the top layer gossip protocols are set with the same parameter). This means that 20 or 50 neighbors' contact information and profiles are exchanged upon gossip. Figure 2.6 conveys the fact that exchanging more information leads to a faster convergence. After 100 cycles, users have almost converged to their ideal personal networks. However, even though a large number of users are able to discover their ideal neighbors quickly, the remaining users may take a long time to converge. Through intensive analysis of user profiles, those poorly performing users turn out to have few neighbors reflecting their sparse interests. This situation remains marginal.

To evaluate the quality of the top-k results, we run top-10 processing in a centralized implementation of *Exact* and take the 10 returned items for each query as relevant items. This was compared to the results obtained with P3K (unlimited personal network). Figure 2.7 depicts the evolution of R_{10} for three values of recall as personal networks converge. A recall of 1 means that the same results as the centralized solution are achieved.

At cycle 50, more than 77% of the queries get exactly the same results as in the centralized approach and the rest of the queries retrieve at least 8 out of 10 relevant items. R_{10} keeps improving with time and about 98.5% of the queries obtain all relevant items at cycle 250. This evaluation shows that P3K, with unlimited personal networks, provides almost the same personalized top-k results as in the centralized approach.

P3K personal network evaluation (limited) We measure, in the dataset used for the experiments, the average size of unlimited personal networks, referred to as *networkSize*, and the maximum *networkSize* over all users. The results are reported in Table 2.2. On average, each user maintains about 18% other users as neighbors regardless of the size of the system. This is clearly too high a

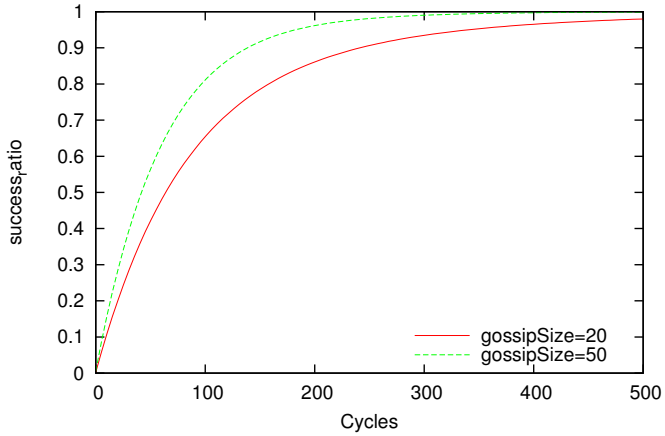


Figure 2.6: Personal network convergence

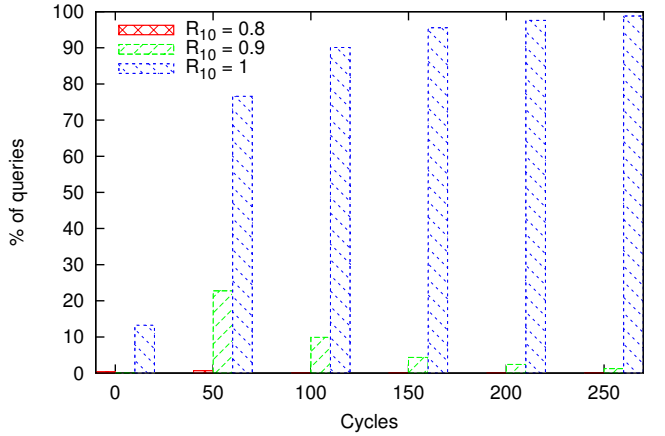


Figure 2.7: Recall evolution (gossipSize=50)

number when considering large-scale networks. We now evaluate the impact of limiting the size of personal networks.

Table 2.2: *networkSize* in systems of different scale

Number of users	Average *networkSize*	Max *networkSize*
1000	180	750
5000	897	4165
10000	1801	8350

A limited number of the s most representative neighbors are now maintained in the personal networks. Again the baseline reference for the top-k results are the ones achieved by the centralized

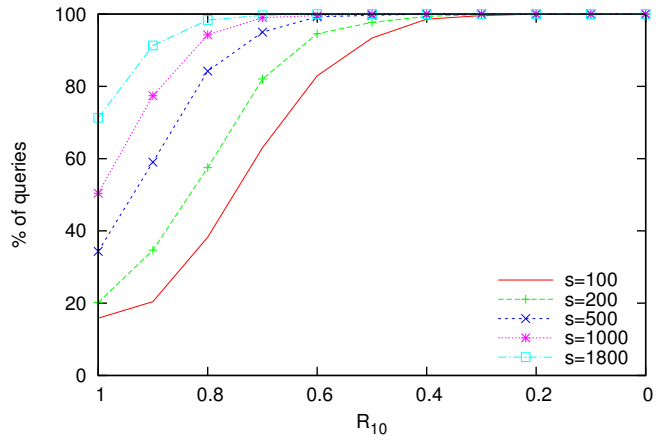


Figure 2.8: Effect of personal network size (s) on recall

implementation, which we compare now against our decentralized solution with the size of personal network restricted to s (for various values of s). We also compare the selection procedure underlying P3K (the s users with the highest similarity as defined by Formula 2.2) with three alternatives:

Random. At each cycle, only s random users are kept as neighbors if more than s users meet the unlimited personal network criterion. The intuition is that a random sample is usually representative of the population from which it is drawn. However, this sampling is not uniform as in-degree of each peer are not limited while constructing the overlay network. Therefore, users having high in-degrees are more likely to be selected as neighbors.

Biased Random. This is a probabilistic version of our P3K selection procedure: s users are selected as neighbors. The probability that a user u_j is kept as neighbor of user u_i is proportional to the similarity between them. So users with higher similarity are more likely to be chosen as neighbors while users with lower similarity are not dropped completely to avoid losing the variety of items in the later constructed inverted lists.

Nearest. As users are more likely to benefit from users having similar interests, measured by the similarity between them, this strategy has complete confidence in users possessing most similarity with the gossip initiator and chooses them as neighbors.

Cumulative curves in Figure 2.8 show the percentage of top-10 queries achieving a given recall (X-axis) for different sizes of personal networks. Not surprisingly, as only a limited number of neighbors are kept in personal networks, the recall fails to attain 1 for all queries. However, more than 80% of the queries can get at least 7 relevant items with at most 200 neighbors, which is only 10% of the average *networkSize*.

We consider a recall larger or equal to 0.7 as satisfactory. We ignore in Figure 2.9 the users with a personal network of size less than the one considered in the X-axis. For those, the limitation is not meaningful as they have their complete personal networks. P3K outperforms the alternative approaches in terms of top-10. When *networkSize* is relatively small, the difference among these strategies is prominent, while the discrepancies are less visible for higher values of *networkSize*. The difference and similarity among these strategies also show that users with similar tagging behaviors are more representative and contribute more to the final top-k results.

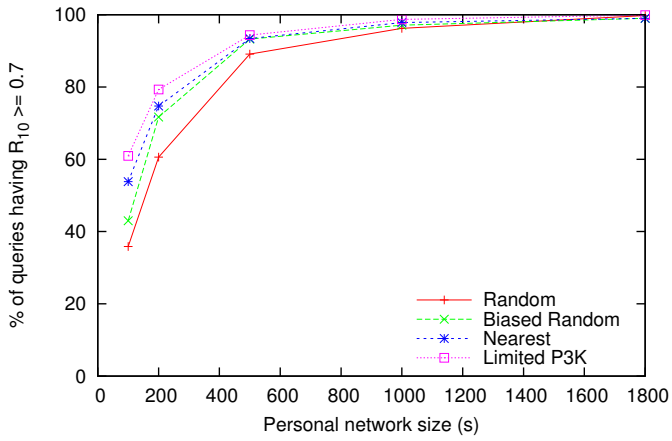
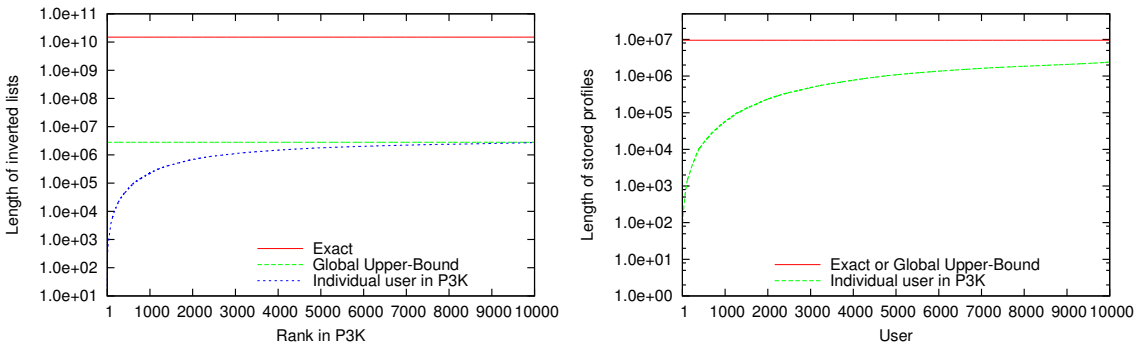


Figure 2.9: Recall of different strategies

2.3.3.3 Quantitative P3K evaluation

Space requirement Not surprisingly, the maximum space requirement, over the centralized approach, is reached when the personal network is unlimited. Figure 2.10 compares individual user's space requirement with that in *Exact* and *Global Upper-Bound* in [12].

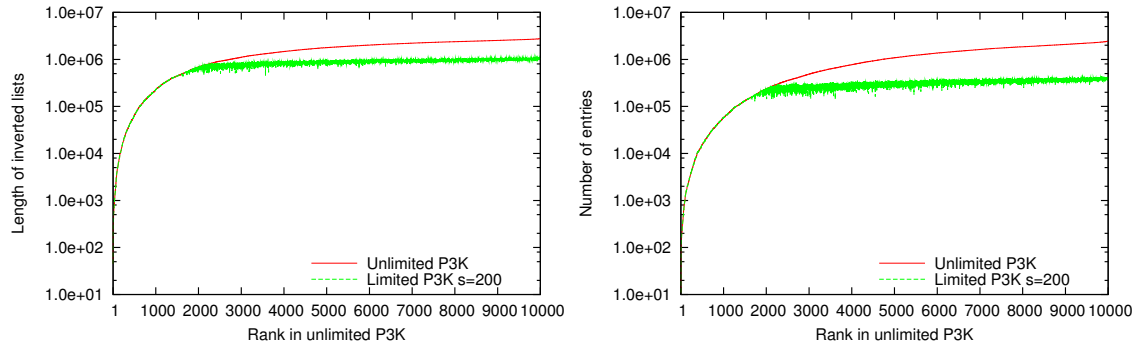
The space required by a user depends on the number of tagging actions involved in her neighbors' profiles (Figure 2.10(a)) and the total length of the inverted lists (Figure 2.10(b)). Users are ranked in ascending order of their space requirements for unlimited personal network and the value on the X-axis can be considered the rank. As expected, no user needs to store as much information as in the *Exact* reference approach of [12], nor even as the *Global Upper-Bound* approach, the most effective storage wise among those in [12]. Moreover, the user with the largest neighbors' profiles stores only 20% of the whole tagging action set.



(a) Length of stored inverted lists

(b) Length of stored user profiles

Figure 2.10: Space requirement in unlimited personal network



(a) Length of inverted lists

(b) Length of stored user profiles

Figure 2.11: Space requirement in limited P3K personal network

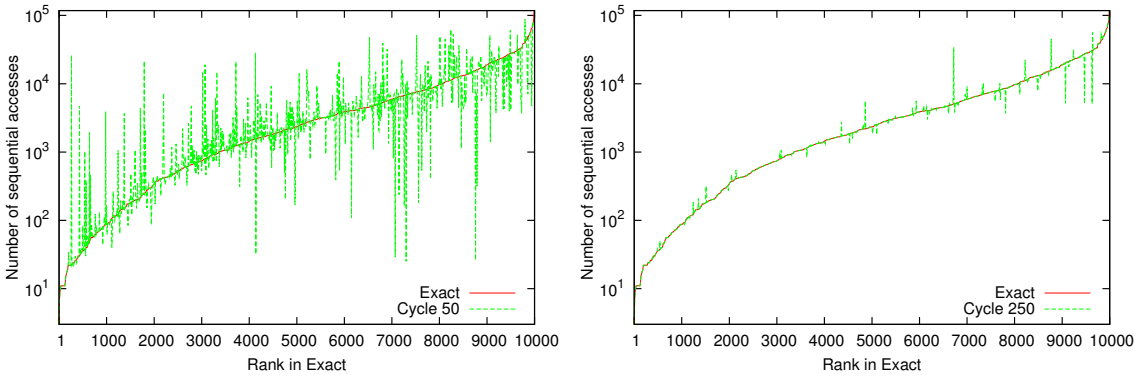
Although no user needs to maintain the entire set of profiles as a centralized approach would do, keeping 20% of that set in a personal hard drive might reveal critical. Figure 2.11 shows however how the space requirement can be significantly limited: 18.2% of the users have a personal network of size less than 200. This corresponds to the part of the curves that overlap in both unlimited and limited versions of P3K. For the others, 49.1% of the entries in inverted lists and 69.2% of the entries in neighbors' profiles are saved. Hence, the space clearly becomes less of an issue in a fully decentralized approach such as P3K.

Query response time Figure 2.12(a) illustrates the response time at gossip cycle 50 when users find on average 90% of their ideal neighbors in unlimited personal networks. As the query is processed locally and there is no transmission delay, the processing time is measured by the number of sequential accesses in inverted lists. Here, we compare only the processing time with that of *Exact*, shown to be the most efficient among the algorithms in [12].

Interestingly, shorter inverted lists do not necessarily mean less execution time because the lack of information may decrease the scores of certain items and, as a result, more entries in the lists may need to be checked to get the final top-10 items. However, the penalty in time consumption remains reasonable. On average, only 3% more time is required to process these queries. If the network continues to converge, at cycle 250, only a handful of queries exhibit different processing time as with the querier's ideal personal network (Figure 2.12(b)).

For P3K, as inverted lists are changed, the number of sequential accesses to get the top-10 items is no longer the same as before (Figure 2.13). About 18.2% of the queries require the same time to retrieve the results while 39.9% consume less time and 41.9% require more time. On average, only 5% more sequential accesses are required for each query.

Scalability As shown in Table 2.2, the size of unlimited personal networks grows linearly with the number of users in the system. With P3K, we observe from Figure 2.14 that the necessary number of neighbors to obtain the top-10 results of the same quality remains stable even with an increasing number of users. Clearly, the larger the network, the smaller the ratio of necessary neighbors in the



(a) Number of sequential accesses (Cycle 50)

(b) Number of sequential accesses (Cycle 250)

Figure 2.12: Number of sequential accesses in unlimited personal network

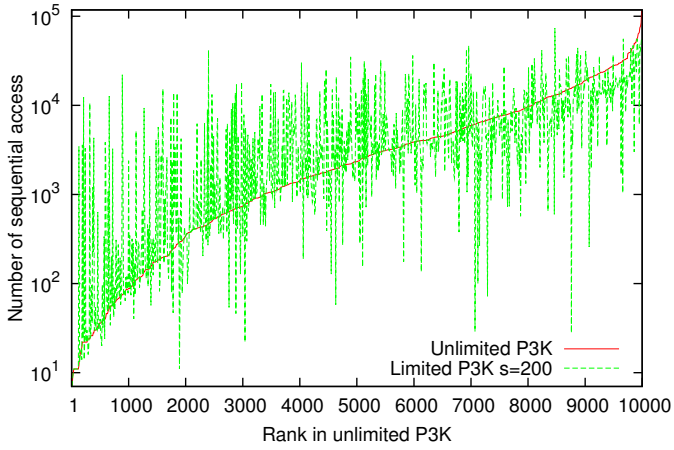


Figure 2.13: Number of sequential accesses in limited P3K personal network

personal network (s) over the total number of users (N). This shows that P3K scales well: there is no need to maintain large personal networks to obtain good personalized top-k results. Well selected neighbors, i.e., those sharing the most similar tagging behaviors, bring most of the relevant items and preserve their relative order to a query even though their overall scores decrease.

2.3.4 Summary

We presented in this section our first peer-to-peer approach to personalize top-k processing in collaborative tagging systems. We described the design and implementation of our approach and investigated its performance. The evaluation shows that P3K *(i)* provides comparable top-k results to the state-of-the-art centralized approaches [12]; *(ii)* saves storage space by limiting the size of users' personal networks; *(iii)* exhibits a slight overhead, with respect to computation time, over

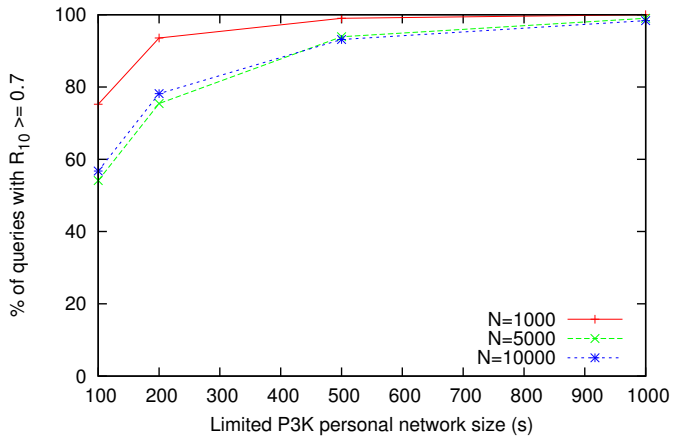


Figure 2.14: Recall in systems of different scale

the fastest centralized variant in [12]. The experimental results are encouraging and we believe that decentralization is the right way to provide personalized top-k processing in a scalable manner.

Still, it leaves several research issues. First of all, P3K achieves scalable top-k processing by limiting the size of the personal network. Although this ensures a good approximation of the results that can be obtained in the reference centralized system, the required storage may still be prohibitive for certain users using mobile devices with very limited capability. In fact, better results should be expected if more information beyond the limited personal network is accessible during the query processing. Moreover, maintaining the personal network requires measuring the similarity between users. Gossiping the whole profiles of the users may incur unnecessary bandwidth consumption as only the users who are similar enough to a user are kept in her personal network. In addition, users in collaborative tagging systems are very active and frequently tag new items. Any change in a user's profile may trigger the evolution of her personal network, as well as the query results. Actually, a robust system should be able to follow such changes timely. We try to address these issues in the next section.

2.4 Gossiping personalized queries

As we have seen, our fully decentralized solution P3K, consists for each user to locally store and maintain her implicit social network, enabling thereby efficient top-k query computation while alleviating the scalability problem faced by the state-of-the-art centralized solutions [12]. Yet, P3K requires every user to store all profiles of the acquaintances in her personal network: these are then massively replicated and hence hard to maintain. As shown in Figure 2.8, 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 consists for each user to store and maintain only her own profile and seek other profiles on the fly, i.e., 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

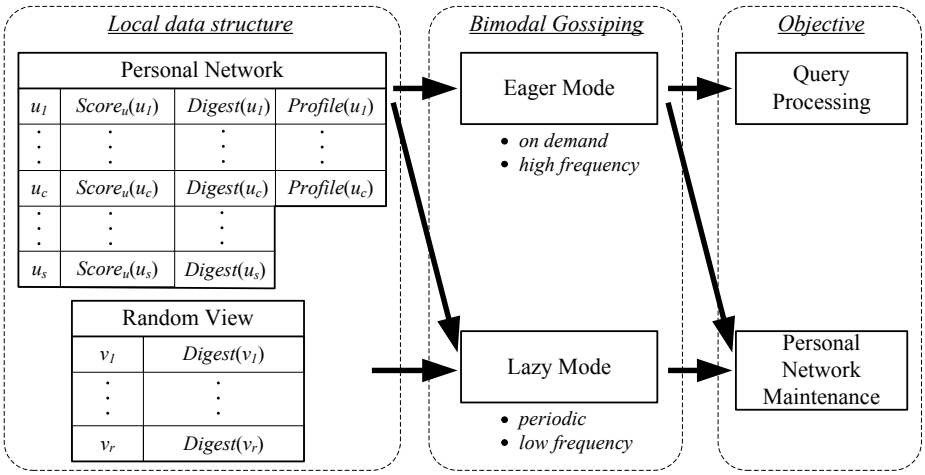


Figure 2.15: System model

of temporarily disconnected users may be unavailable which, in turn, might significantly hamper the accuracy of the query processing.

In this section, we present P4Q (*Performant 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 tagging profiles. Each user however only locally stores a limited subset of profiles according to her storage capability. Different from P3K, where the query is locally processed by the queries, the query in P4Q is gossiped and computed collaboratively. In the following, we explain in detail *(i)* how the personal networks are maintained with respect to the storage capability and bandwidth requirement; *(ii)* how the queries are gossiped to avoid saturating the network by contacting all the users in the personal network at the same time, and refresh the part of the network originating from the querier by generating a wave of refreshments in the personalization process; *(iii)* how the users react to dynamics involved in the system.

2.4.1 System model and data structures

As in P3K, each P4Q user maintains two data structures: a personal network and a random view. Yet, as shown in Figure 2.15, the specific data maintained in the personal network and the random view are slightly different. For the random view, only a digest of profile is stored for each neighbor.

Personal network The personal network of a user $u_i \in U$ is a set of s neighbors having the most similar interests with her, noted as $Network(u_i)$. A similarity score $Similarity(u_i, u_j)$ is used to quantify the similarity between the user u_i and her neighbor u_j. We define the similarity score as the number of common tagging actions in two users' profiles, i.e.,

$$\begin{aligned} Similarity(u_i, u_j) &= |Profile(u_i) \cap Profile(u_j)| \\ &= |\{\langle i, t\rangle | Tagged(u_i, i, t) \wedge Tagged(u_j, i, t)\}| \end{aligned}$$

Here we use the term "*common tagging action*" to stand for the couple $\langle i, t\rangle$, where the same item i was tagged by both u_i and u_j with the same tag t.

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 (tags) and specific objects (items) into account. The higher the score, the more interests are shared between u_i and u_j. This definition is consistent with what we used in Section 2.3.1.2 to limit the size of P3K personal network and was proven to be effective. The only difference is in P4Q the similarity is computed directly with this metric without passing through Formula 2.1. In fact, the similarity score is application-specific and P4Q is independent of the way similarity is defined.

The query results of a user u_i 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_j having the highest $Similarity(u_i, u_j)$ 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 availability. Typically, the larger c, the more accurate the results obtained by computed the query locally.

In order to limit the overhead of the protocol, a digest of profile ($Digest(u_j)$) is also stored along with each neighbor in the personal network. A digest is a compact summary of a user's profile encoded using a Bloom filter [108]. The digests are used in P4Q to estimate the similarity between users.

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. A Bloom filter [108] 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, *i.e.,* 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 filter to assess the similarity score between users encountered during the personal network maintenance. As a consequence, using Bloom filters, P4Q may over-estimate a score due to the false positive answers. Yet, should 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 2.4.2.1. 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 un 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.

2.4.2 Bimodal gossiping

P4Q relies on a two-mode gossip protocol as represented in Figure 2.15. 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. We now describe the lazy and eager modes of the P4Q gossip protocol.

2.4.2.1 Maintaining personal networks: The lazy mode

The personal network of each user is discovered and maintained through a two-layer gossip as described in Section 2.3.1.1. The mere difference is that only the profile digests are exchanged between users in the bottom layer.

As we know, the gossip message of a user in the top layer is composed of a subset of her neighbors' profiles. In P4Q, these profiles are randomly selected from the c profiles stored in the user's personal network. During a gossip between the user u_i and the u_j, they first send to each other a such message. Then u_i computes $Similarity(u_i, u_l)$ for each received user u_l and $Similarity(u_i, v_j)$ for each user v_j in her random view. If v_j qualifies to be incorporated in the personal network, the profile of v_j is obtained by directly contacting v_j. 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.

To avoid overloading the system, the transmission of profiles in the top layer gossip follows a 3-step protocol. Algorithm 2.2 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_l to become a neighbor of the user u_i only if *(i)* this estimated upper-bound is larger than the similarity score between u_i and the least similar neighbor in her personal network, or *(ii)* this upper-bound is larger than 0 while the user u_i has not yet the desired number of neighbors in her personal network. Otherwise, there is no need to exchange the profile since u_l can not qualify for the personal network of u_i.

Algorithm 2.2 Gossiping profiles in lazy mode

1. **Input:** $Profile(u_i)$ & received profile digests
2. **Output:** new $Network(u_i)$
3. **for** each received $Digest(u_l)$ **do**
4. **if** $u_l \in Network(u_i)$ **then**
5. **if** $Digest(u_l)$ does not change **then**
6. drop $Digest(u_l)$
7. **else**
8. estimate the similarity upper-bound $UbScore(u_i, u_l)$
9. **if** $|Network(u_i)| < s$ and $UbScore(u_i, u_l)$ **then**
10. add u_l to $Candidates$
11. **else if** $| Network(u_i) | = s$ **then**
12. **if** $UbScore(u_i, u_l) > Min\{Similarity(u_i, u_m), u_m \in Network(u_i)\}$ **then**
13. add u_l 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_l in $Candidates$ **do**
21. require her tagging actions for computing $Similarity(u_i, u_l)$
22. **end for**
23. receive the required information
24. **for** each u_l in $Candidates$ **do**
25. compute $Similarity(u_i, u_l)$
26. **if** $Similarity(u_i, u_l)$ is one of the s highest scores **then**
27. add u_l to $Network(u_i)$ with $Similarity(u_i, u_l)$ and $Digest(u_l)$
28. **end if**
29. **end for**
30. **for** each u_l added to $Network(u_i)$ **do**
31. **if** $Similarity(u_i, u_l)$ is one of the c highest scores **then**
32. require the rest of the tagging actions in $Profile(u_l)$
33. **end if**
34. **end for**
35. **end if**

- Exact score computation: The second exchange (19-29) of tagging actions that contribute to the upper-bound estimation enables to pre-compute 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 neighbors' profiles.

- Profile exchange: The last exchange (30-35) only happens if there are indeed profiles that should be stored, namely if u_l is one the c most similar users for u_i.

Figure 2.16 further illustrates the upper-bound estimation and the exact score computation during a gossip through an example.

We now detail how the upper-bound of similarity score is estimated based on the *ItemTagBloom* digest. Both a Bloom filter of items and a Bloom filter of tags are used to form a user's profile

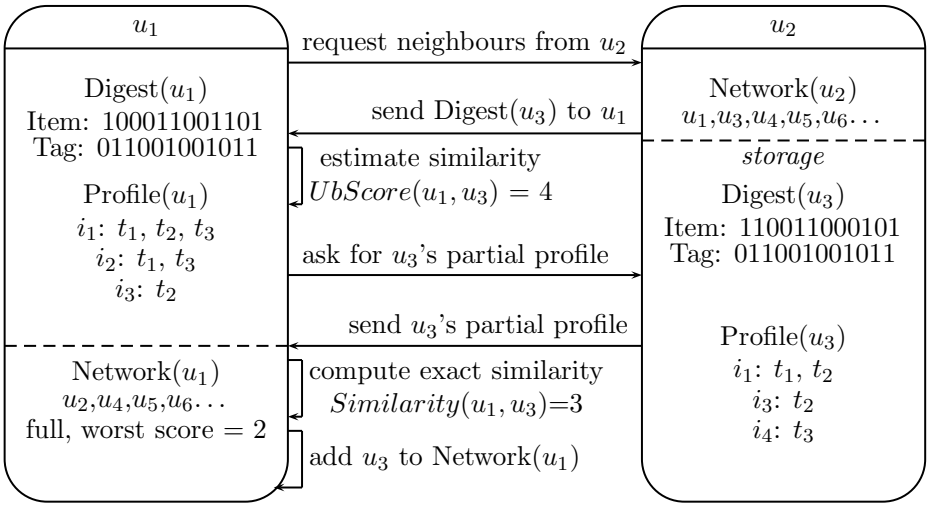


Figure 2.16: Estimating scores in gossip

digest. When the user u_i receives the profile digest of the user u_l, 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 to tag this item is also used by u_l. The score upper-bound can thus be estimated by counting her tagging actions where both the concerned items and tags are used by u_l. More formally, the score upper-bound can be expressed as

$$UbScore(u_i, u_l) = |\{\langle i, t\rangle | Tagged(u_i, i, t) \wedge Tagged(u_l, i, *) \wedge Tagged(u_l, *, t)\}|,$$

where $Tagged(u_l, i, *)$ represents an item tagged by the user u_l regardless of the tags used and $Tagged(u_l, *, t)$ represents a tag used by u_l in her profile regardless of the tagged item.

Specifically, as shown in Figure 2.16, when the user u_1 receives the profile digest of the 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 $Digest(u_3)$. Then u_1 queries her tags t_1, t_2 and t_3 of i_1 in the tag part of $Digest(u_3)$ 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.

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 re-generation of the corresponding profile digest. Note that the false positive would not lead to any incorrectness while selecting the neighbors for the personal network since a misjudgment only over-estimates 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 the 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.

2.4.2.2 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 similarity 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. Firstly, it is unlikely that the profiles stored by random neighbors 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 [26].

The eager mode of P4Q works as follows and the first cycle of gossip is illustrated in Figure 2.17. The querier u_i 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 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 but would contribute to the query processing. While the query is collaboratively answered by potentially all the neighbors in u_i's personal network, only the neighbors who have used the tags in the query are actually involved in the query processing. The remaining list of the user u_i 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 not locally stored by u_i (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 is missing. As we will see, the profiles that can not 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.

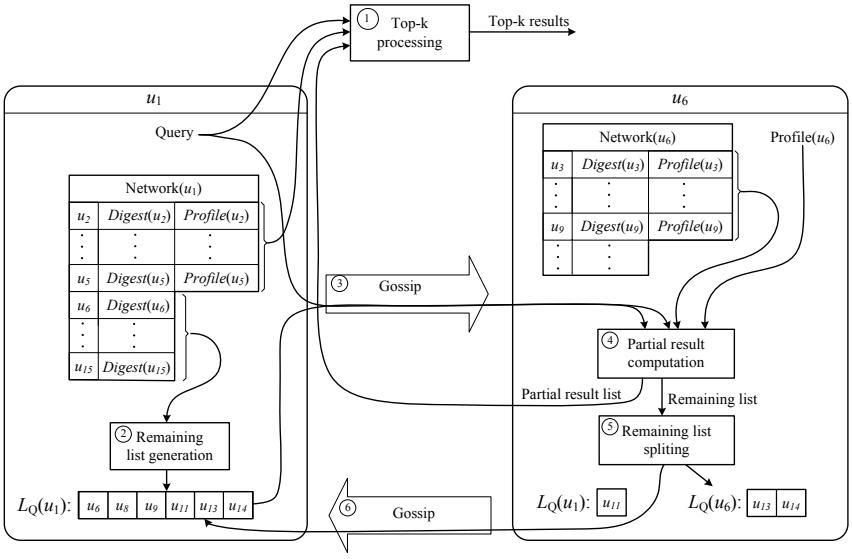


Figure 2.17: Query processing in eager mode (1st cycle)

Those profiles of users in the remaining list are discovered through gossip. User u_i initiates a gossip with a neighbor u_j having the oldest timestamp in $L_Q(u_i)$ and attaches the query and the remaining list to the gossip message containing the profile digests she wants to send to u_j as described in the lazy mode (③). 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 (④). 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: $L_Q(u_j)$. 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 other of her neighbors. 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 (⑦).

In the second cycle, both u_i and u_j 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 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 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 2.3 is the query processing at the querier, whereas Algorithm 2.4 shows how a query is gossiped between two users. The gossip initiator (u_init) is the user who forwards the query and the remaining list and the gossip destination (u_dest) 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 2.4.6.4).

Algorithm 2.3 Query processing at the querier

1. **Input:** querier u_i's query Q & $Network(u_i)$
2. **Output:** personalized query results of Q
3. process Q with the profiles in $Network(u_i)$
4. **if** u_i stores all her neighbors profiles **then**
5. display query results and return
6. **else**
7. build the remaining list $L_Q(u_i)$
8. **repeat**
9. gossip with a neighbor u_j in $L_Q(u_i)$
10. receive new $L_Q(u_i)$ from u_j
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**

Algorithm 2.4 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** $\exists u_l \in L_Q(u_{init})$ & $u_l \in Network(u_{init})$ **then**
5. $u_{dest} \leftarrow u_l$ 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. remove each u_l from $L_Q(u_{init})$ if $Profile(u_l)$ is stored by u_{dest}
19. $L_Q(u_{dest}) \leftarrow (1-\alpha)$*$|L_Q(u_{init})|$ users from $L_Q(u_{init})$
20. $L_Q(u_{init}) \leftarrow L_Q(u_{init}) \backslash L_Q(u_{dest})$
21. send $L_Q(u_{init})$ to u_{init} in gossip message
22. process Q with profiles required by u_{init} and stored by u_{dest}
23. send partial result to the querier
24. maintain personal network as in lazy mode
25. **end loop**

2.4.3 Collaborative top-k query processing

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

Queries and scoring We consider a query $Q = \{t_1, ..., t_n\}$, 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. We always consider the same scoring function as in [12]. 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.,

$$Score_{u_j,Q}(i) = |\{t_m | t_m \in Q, Tagged(u_j, i, t_m)\}|.$$

Thus 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 neighbor u_j in u_i's personal network, i.e.,

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

Similarly, alternative monotonic scoring functions can also be used to compute the relevance score.

Top-k processing in P4Q As presented above, 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 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 relevance 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 Accesses) [47] algorithm to P4Q while minimizing the processing time. In P4Q, at the end of each cycle, k items are returned to the querier. Algorithm 2.5 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

Algorithm 2.5 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. $ScanningLists \leftarrow$ new partial result lists
4. $ScanningPosition \leftarrow 1$
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** $i \in$ 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**

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.

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. These consist of part of the future work.

2.4.4 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 2.4.2.2, at least 1 profile can be found ($X \geq 1$) by contacting its owner in the gossip.

Theorem 2.4.1. *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 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 $(1-\alpha)(l-X)$. α is the splitting factor as described in Section 2.4.2.2. 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_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

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

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

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

If $0 \leq \alpha < 0.5$, 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 2.4.2. *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 monotonically increasing with α if $0.5 \leq \alpha < 1$ and monotonically 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 $L \geq X$ 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 monotonically increasing with α if $0.5 \leq \alpha < 1$.
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 monotonically 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 2.4.3. *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, i.e., their remaining lists become 0 at the same cycle. Then at the 1st 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 gossip 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 2.4.4. *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 1st cycle, one gossip is done between the querier and one of her neighbor. Supposing all the users involved in the processing finish their remaining lists at the same time, at the 2nd cycle both the querier and her neighbor 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 the other 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. □

2.4.5 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 and potentially changes 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 to proactively disseminate her profile upon changes. In the standard algorithm, u_i picks u_j with the oldest timestamp to gossip with in her personal network. The neighbor relation in P4Q is not necessarily symmetric and u_j may not consider u_i as her neighbor. Therefore the changes in u_i' s profile might not be immediately taken into account, until u_i encounters a similar neighbor.

To address this issue, when u_i changes her profile, she proactively promotes her new profile by picking from her personal network the most similar user u_l instead of the user u_j with the oldest timestamp. The intuition is that even if the neighbor relation is not symmetric, as u_l has the highest similarity score, she is also the most likely user to store u_i's profile. Self-promotion helps

in disseminating the new profile, starting with the users who are the most likely to be interested in u_i'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 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 2.4.2.2, 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), 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, she will gossip the up-to-date version of that profile regardless of the subset selection.

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

2.4.6 Experimental evaluation

In this section, we report on the evaluation of P4Q, which has also been conducted using *PeerSim*. The same delicious dataset and queries as those for evaluating P3K are used. We first assess in Section 2.4.6.2 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 2.4.6.3. We finally evaluate in Section 2.4.6.4 the ability of P4Q to deal with profile changes and user departures.

2.4.6.1 Experimental setup

System setting As defined in Section 2.4.1, 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 1000 in our simulations. In fact, regardless of the size of personal network, the querier can get the accurate results within a limited number of cycles (Theorem 2.4.1). As mentioned above, 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 1000 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 2.3. 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.

Table 2.3: Distribution of stored profiles (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%

Profile digest selection As described in Section 2.4.2.1, 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 later case, the exact similarity is computed if there is any common item between two profile digests of *ItemBloom*. The evaluation shows that *ItemTagBloom (i)* outperforms the third alternative 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_i first checks if the items from her profile are encoded in u_l's digest. The score upper-bound is estimated by u_i as the number of her tagging actions related to the common items with u_l, contained in her digest, i.e.,

$$UbScore(u_i, u_l) = |\{\langle i, t\rangle | Tagged(u_i, i, t) \wedge Tagged(u_l, i, *)\}|.$$

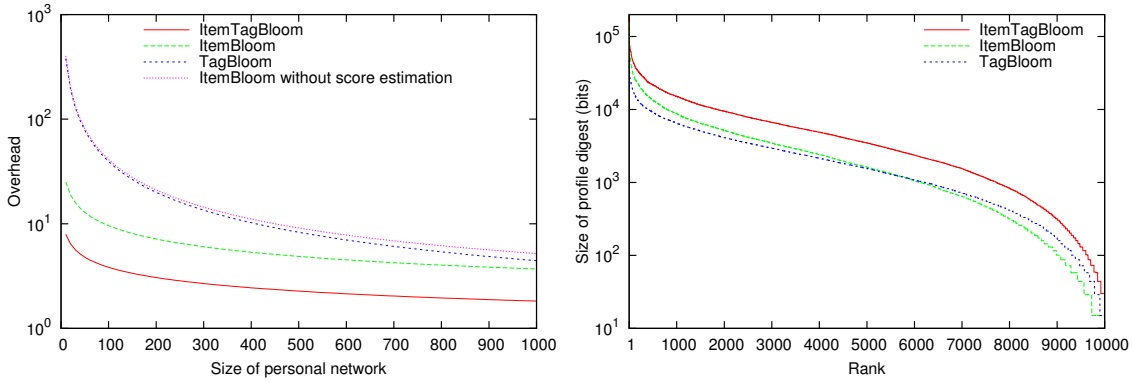
  Only the tagging actions involved in the computation of score upper-bound contribute to the exact similarity score.

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

$$UbScore(u_i, u_l) = |\{\langle i, t\rangle | Tagged(u_i, i, t) \wedge Tagged(u_l, *, t)\}|.$$

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

Figure 2.18 compares the computation overhead and the potentially 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



(a) Accuracy of similarity upper-bound estimation

(b) Size of profile digests

Figure 2.18: Comparison of profile digests

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 2.4.2.1. The larger the overhead, the less accurate the estimation of the similarity upper-bound is.

The average computation overhead for different sizes of personal network is shown in Figure 2.18(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 always requires the most computation. With a personal network of 1000 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 2.18(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 above 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, 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 (23.6K bytes) and the average size is only $6,277$ bits (0.78K bytes). 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 best balance between the accuracy of estimation and the size of profile digest.

Figure 2.19 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 the size of the digest by a large factor

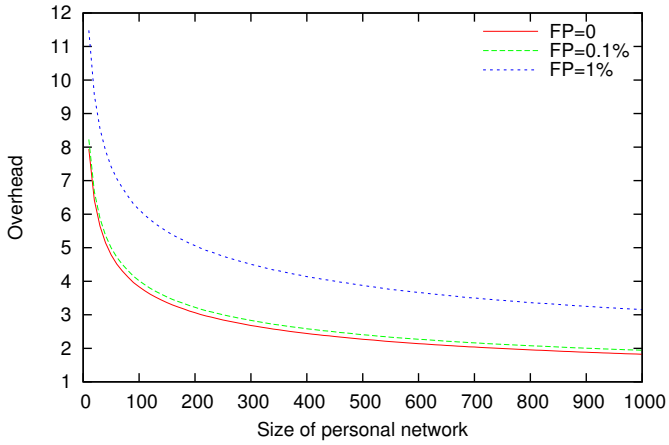


Figure 2.19: Overhead of *ItemTagBloom* with different false positive rates

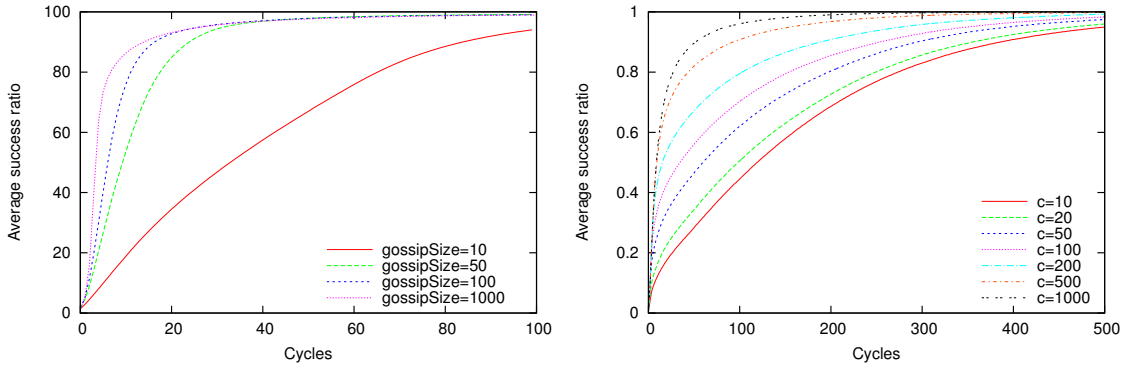
depending on the size of an item identifier or a tag (12.4 in our dataset). Therefore, we use a false positive rate of 0.1% in the following.

2.4.6.2 Qualitative P4Q evaluation

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 average success ratio as defined in Section 2.3.3.1.

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 2.20(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.

Figure 2.20(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 200th 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.



(a) Gossip sizes

(b) Stored profiles

Figure 2.20: Personal network convergence

Query processing in eager mode As described above, 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. 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 P4Q achieve the same top-k results as the baseline.

Figure 2.21 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

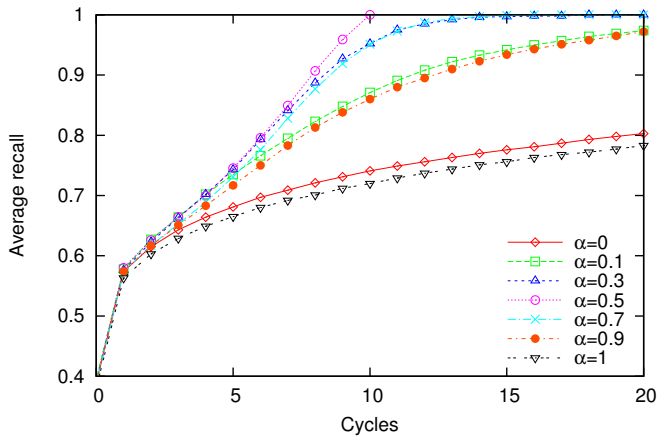


Figure 2.21: Impact of splitting factor (α) on average recall ($c = 10$)

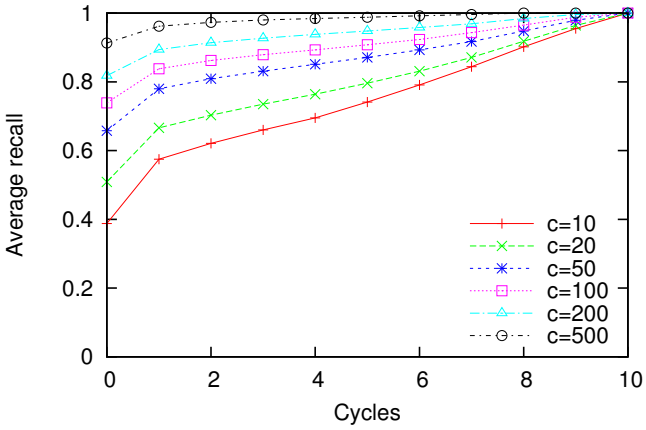


Figure 2.22: Impact of stored profiles (c) on average recall ($\alpha = 0.5$)

P2P system [106]. 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 2.21).

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 2.21 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.

Figure 2.22 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 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 2.4.6.4).

2.4.6.3 Cost analysis

Storage requirements As opposed to P3K where the each user stores all their neighbors' 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 0.78K bytes. The total storage required for the profile digests (1000 in personal network and 10

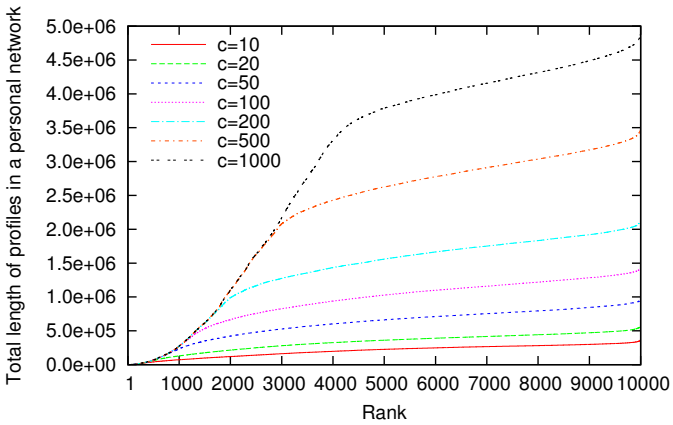


Figure 2.23: Space requirement

in random view) is thus 787.8K bytes.

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 still use the metric in Section 2.3.3.1 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 2.23 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 their ranks. 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 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 12.5M bytes. This requirement can even be fulfilled by mobile devices with limited capabilities.

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 $\lambda = 4$.

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 39K bytes for each user. Only 7.8K and 15.6K bytes are transmitted for users having 10 or 20 profiles in their personal networks.

Except for the profile digests, the information received 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.

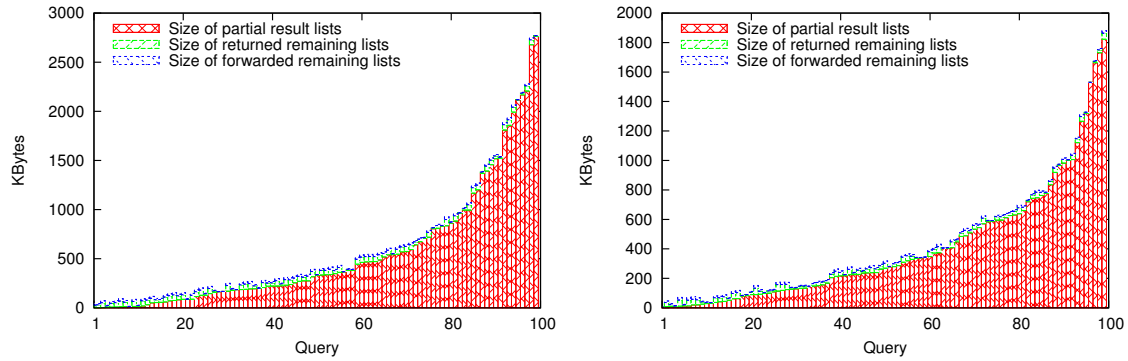
In the $\lambda = 1$ scenario, on average, at each cycle, before the personal networks stabilize, 74.6% of users in the system have to transmit further information for measuring the exact similarity while only 4.5% of them require the exchange of the whole profiles. For these users, 8.94K bytes and 528K bytes are transmitted respectively if they have such need in a give 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, 79.6% users have to transmit 12.56K bytes for measuring the similarity while 15.3% of them need the whole profiles of 580K bytes. 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 capability. In the bottom layer of the lazy gossip, 10 profile digests of 7.8K bytes are exchanged at each cycle. In fact, using the profile digests to estimate the similarity score upper-bounds avoids 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. This is due to the fact that unqualified users are immediately pruned after the upper-bound estimation. This brings a save of 6.91K bytes ($\lambda = 1$) and 11.12K bytes ($\lambda = 4$) per cycle for each user.

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 2.24 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, 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 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% its length if no such information is available in both $\lambda = 1$ and $\lambda = 4$ scenarios. 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



(a) $\lambda = 1$ (b) $\lambda = 4$

Figure 2.24: Query processing bandwidth

fact very small. This also verifies the bound on the number of partial result lists of the analysis (Theorem 2.4.3).

2.4.6.4 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 on-line 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.

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 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 2.20(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 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 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, i.e.,

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

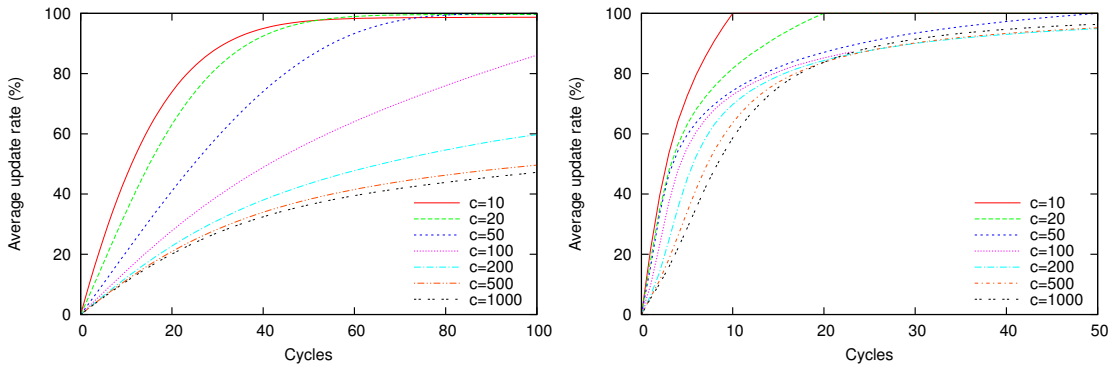
AUR attains 100% when the profiles in all users' personal networks are up-to-date.

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 have 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 2.4 summarizes the influence of profile changes in different settings.

Table 2.4: Impact 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

In lazy mode, i.e., 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. We observe from Figure 2.25(a) that a small number of stored profiles (c) guarantee a high average update rate. After



(a) Reference (b) P4Q

Figure 2.25: AUR evolution in lazy mode with uniform stored profiles (c)

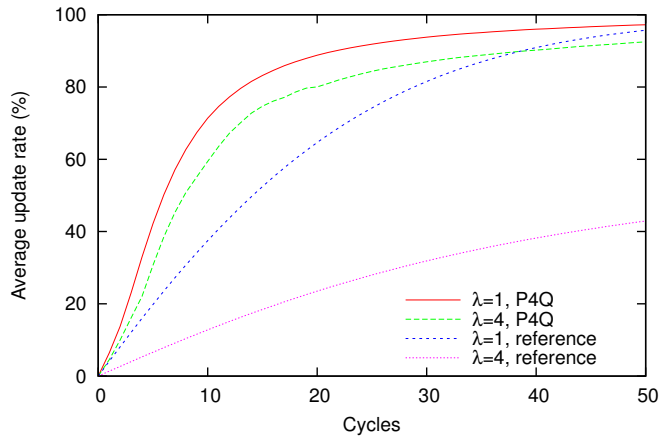


Figure 2.26: Impact of stored profiles (c) distribution on AUR in lazy mode

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. Not surprisingly, the more profiles are stored in the 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 2.25(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. Encouragingly, even if each user stores 500 or 1000 profiles, more than 80% profiles are updated within 20 cycles. This significant improvement in updating rate comes from the fact that users are more voluntary 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 2.26 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, i.e., 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 2.27. 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, only a sublist of users who have used at least one tag in the query are gossiped. This leads to a reduction of 37 users

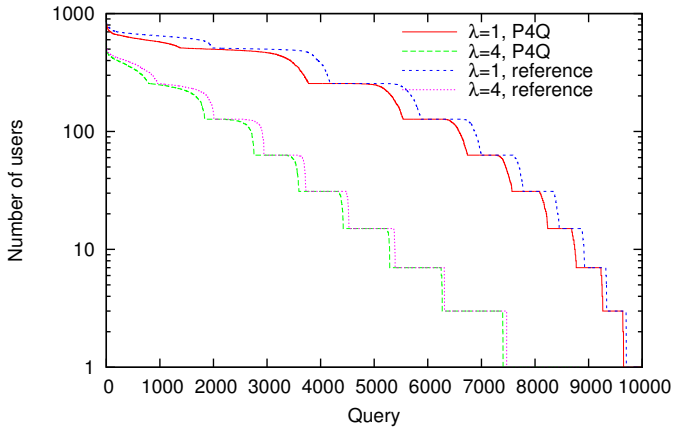


Figure 2.27: Number of users reached by the query

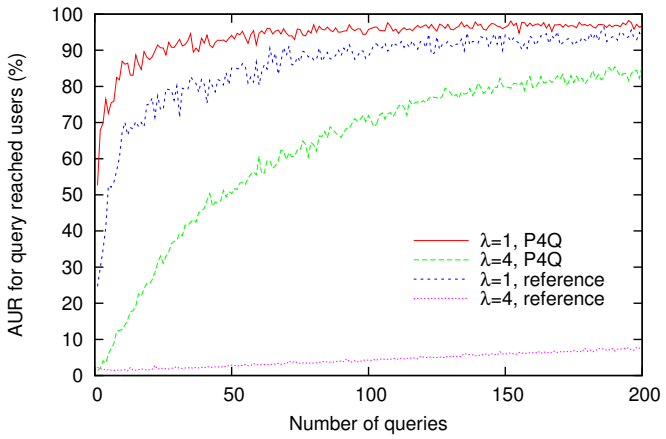


Figure 2.28: AUR evolution in eager mode

($\lambda = 1$) and 9 users ($\lambda = 4$) reached by the query.

Figure 2.28 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 the reference 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. 10 consecutive queries enable all the users reached by the queries to update 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

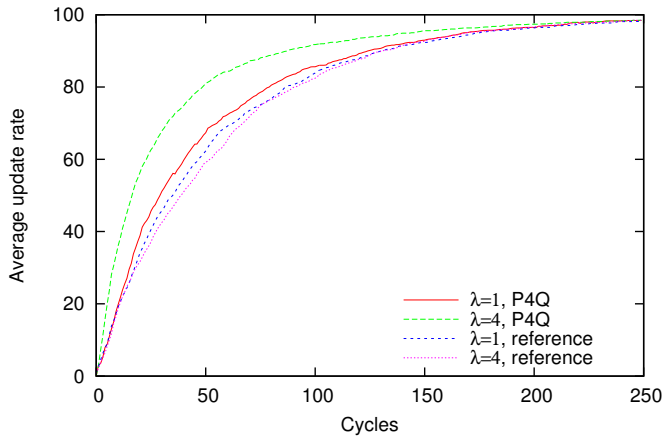


Figure 2.29: Personal network evolution in lazy mode

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 the reference. 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 2.28, is very difficult if users do not actively react to the changes.

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 1719 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 2.29 shows that, if users do not actively react to profile changes (reference), in both settings, after 30 cycles, half of the users have discovered all their relevant neighbors and at the 100th 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 the reference. 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, 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 the reference.

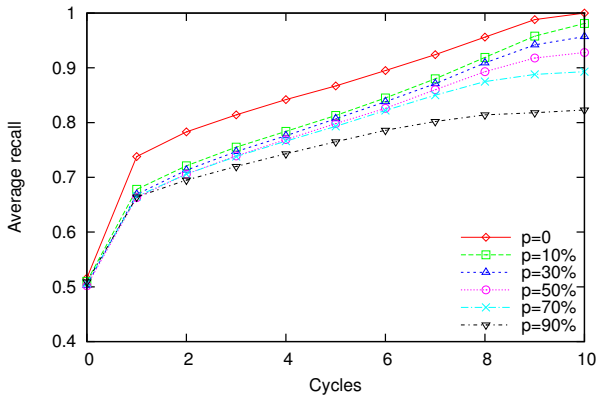
We do not display the results for the eager mode. Effectively the eager mode does not impact the neighbors discovery as the gossip operations are limited to the querier's neighborhood.

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 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.

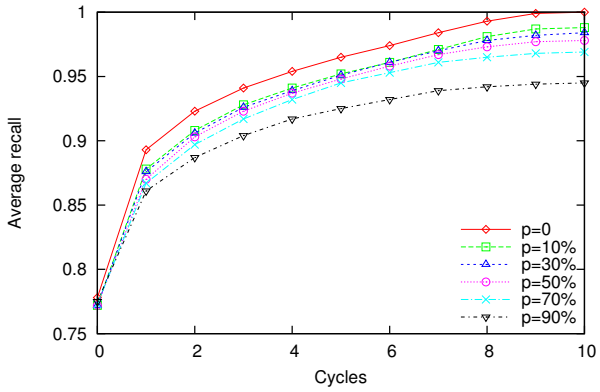
Unfortunately, no information regarding the on-line time of each user could be obtained by crawling a *delicious* trace. So we simply assume that a given percentage of randomly chosen users leave the system simultaneously. Figure 2.30 illustrates the impact of the number of leaving users on the top-k 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 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 2.30(a)). Better results are observed in the $\lambda = 4$ scenario (Figure 2.30(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 2.30(c)). However, even if 50% of the users leave simultaneously in the $\lambda = 4$ scenario, the percentage of non complete 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.

2.4.6.5 Synthesis

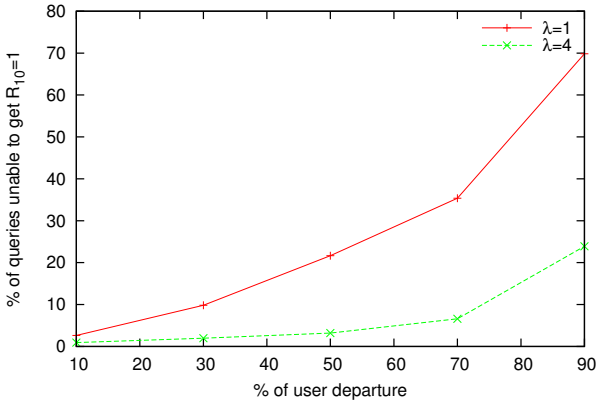
Our evaluation demonstrates that the users get fairly good results immediately. Those results can be refined collaboratively until accurate results are provided within a small number of gossip cycles. Although P4Q takes more time to build the personal networks if the users store less, once most of the neighbors are identified, P4Q guarantees a better freshness of the local stored information and consumes less bandwidth for the personal networks 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 91 Kbps (Kbits per second) for the querier. The background traffic for maintaining the personal network through lazy gossip is only 7.6 Kbps and this may increase to 79.2 Kbps in eager gossip. Even if all users simultaneously change their profiles, in half an hour, 95% of the local stored information is updated and more than 50% users' new neighbors are identified. In fact, if the system that applies P4Q to provide personalized top-k



(a) Average recall evolution ($\lambda = 1$)



(b) Average recall evolution ($\lambda = 4$)



(c) Queries failed to get $R_{10} = 1$

Figure 2.30: Impact of user departure on top-k

processing can tolerate more bandwidth consumption, both the lazy mode and the eager mode can run in higher frequency, which would significantly decrease the personal networks construction time as well as the query processing time.

2.4.7 Summary

In this section, we presented, P4Q, a pragmatic decentralized technique to perform personalized search queries using implicit user affinities. P4Q relies on a lazy mode of gossip to capture such affinities and maintain the personal networks. Profile digests encoded in Bloom filters are used to limit the bandwidth consumption during the gossip of profiles. An eager mode of gossip is used to collaboratively collect the information for query processing so that the users have total freedom to determine the storage they provide with respect to their expectation on query results and response time. As we have seen, the eager mode of gossip also generates a wave of refreshments for the users participating in the query processing. This gives the users clearly incentives to contribute for other users' query processing. In addition, the users in P4Q proactively react to the dynamics involved in their profiles, guaranteeing the robustness and scalability of the query processing in peer-to-peer environments. The efficiency of P4Q was conveyed in both analytical and experimental evaluations.

2.5 Conclusion

In this chapter, we proposed two protocols, P3K and P4Q, to perform personalized query processing in large-scale peer-to-peer systems. Different from the state-of-the-art approaches, the way P3K and P4Q personalize the query processing is inspired by the off-line personalization of [12]. They are however decentralized, which we believe is the key to their scalability.

Both P3K and P4Q rely on a gossip-based protocol to discover and leverage implicit relationship among users. In fact, equipping each P3K or P4Q user with a pre-defined explicit network (e.g., explicit social network in *Facebook*) as input would be straightforward: only the local query processing of P3K or the eager mode of P4Q would suffice.

In fact, P3K can be considered as a special case of P4Q: when the P3K personal network of each user is small, i.e., consisting of only a few neighbors or very small profiles, the whole personal networks can be maintained with the lazy mode of P4Q and stored by each user. As a consequence, the queries can be processed locally without further gossiping. Yet, P4Q provides a more general and complete solution to perform personalized query processing in peer-to-peer systems. 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 alternative techniques could be used.

CHAPTER 3

ON-LINE PERSONALIZED QUERY PROCESSING

3.1 Introduction

The tagging behaviors of users in collaborative tagging systems have been well exploited to enhance the search by personalizing the query results according to the user preferences [67, 103, 63, 66, 12]. As shown in [12], both the processing time and the result quality can be improved by narrowing down the search space within a subset of the system, namely the subset of users having similar tagging behaviors with the querier. The algorithms proposed in Chapter 2 further confirm the efficiency of leveraging the tagging behaviors to personalize the query processing in fully decentralized systems.

Obviously, selecting the right subset of users is the key to the result quality. All these approaches are based on the assumption that the tagging profiles are representative enough to model the user preferences. This allows decoupling the personalization and the query processing by pre-computing the subset of users off-line, which in turn ensures short response latency at query time.

Not surprisingly, if a query is highly correlated to the querier's tagging profile, which is used to model her preferences and select the neighbors, the personalized query results would be more satisfactory than the results obtained without any specification. Nevertheless, if the query is not correlated to the querier's profile, typically representing an emerging interest of the querier, the result quality may degrade. For instance, if a computer scientist majored in peer-to-peer systems dedicates to a new project and wants to find some papers about "database system" and "information retrieval", she may hardly find any valuable information from the limited number of users in her personal network who are solely interested in peer-to-peer systems. Involving more such users in the query processing may increase the probability to find the desired papers but it appears to be not really efficient considering the huge number of scientists working on peer-to-peer systems but not really knowledgeable in the search topics.

Clearly, in order to effectively handle both queries that are correlated with the tagging profile, as well as those that are not correlated to that profile, the adequate personalization should be carried out based on both *(i)* the tagging profile of the querier as well as *(ii)* the query itself. In this way, the

emerging interest can be explicitly expressed while personalizing the query processing. In addition, this would ensure the querier to receive some interdisciplinary papers that aim to build distributed databases in priority for example.

Putting this into practice requires first to refine the user preference by leveraging both the tagging profile and the query. This in turn requires performing the personalization *on-line* because the emerging interests can only be realized when a query is issued. In this chapter, we investigate two algorithms to perform such on-line personalized query processing. The first algorithm, DT^2, is designed for classical collaborative tagging systems, where a central server is in charge of the query processing. The second algorithm, DT^{2}P^2, achieves the same goal in a fully decentralized environment in collaboration of the users in the system. We introduce DT^2 in Section 3.2 and DT^{2}P^2 in Section 3.3 respectively.

3.2 DT^2: centralized on-line personalization

3.2.1 DT^2 in a nutshell

We propose in this section the DT^2 protocol that performs on-line personalized top-k processing in centralized collaborative tagging systems. More specifically, we consider a collaborative tagging system (folksonomy) as $\mathbb{F} = \langle \mathbb{U}, \mathbb{I}, \mathbb{T} \rangle$. A triple $\langle u_o, i_m, t_n \rangle \in \mathbb{F}$ captures the fact that user u_o tagged item i_m with tag t_n. A user u_o's profile is represented by her tagging history, namely all the triples associated with u_o. We denote by $I(u_o) \subset \mathbb{I}$ and $T(u_o) \subset \mathbb{T}$ the sets of items and tags respectively associated with u_o. Users issue queries in the form of keywords (tags). Given a query $Q(u_o) = \{t_1, ..., t_n\}$ made of a set of tags $t_1, ..., t_n$, the goal of top-k processing is to efficiently determine the k items with the highest relevance scores with respect to $Q(u_o)$.

As its name indicates, the DT^2 protocol is itself composition of two underlying top-k protocols as depicted in Figure 3.1.

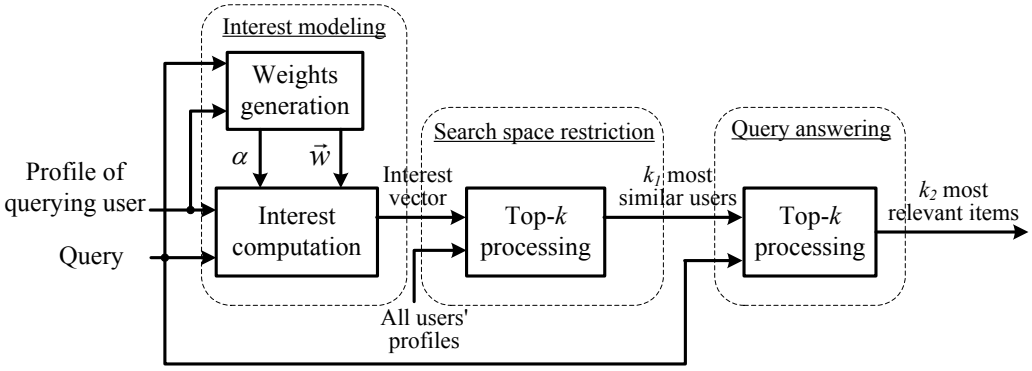


Figure 3.1: Do Top-k Twice (DT^2)

The first top-k protocol underlying DT^2 seeks to determine a small network of appropriate users within which to issue the query, i.e., the personal network of a user for a specific query. The second top-k protocol underlying DT^2 screens the personal network made of these users, and determines the k most appropriate items.

As the query results highly depend on the personal network obtained in the first top-k processing, a crucial technical aspect underlying DT^2 is to model the interest of the querier, at query time, through the computation of a *hybrid interest vector* that accounts for both the tagging profile of the querier and the query itself, while dynamically assigning appropriate *weights* to each. Then the top-k appropriate users can be selected by comparing the querier's hybrid interest against the tagging profiles of other users.

We detail our DT^2 protocol in an incremental manner in the following: we first describe how we derive our notion of querier's hybrid interest from her tagging profile and query; then we introduce our first top-k processing scheme and how we determine, given an interest vector, a personal network; after that, we present our second top-k processing scheme and how we use the personal network above to compute the k most relevant items; at last, we investigate how the size of personal network can be adaptively adjusted for each query to guarantee the best balance between result quality and response time.

To avoid confusions in the following, we denote by k_1 the size of the personal network computed in an incremental manner as we describe in this section and by k_2 the number of resulting items as specified in the original query.

3.2.2 A hybrid interest model

We capture the interest of a user using the classical vector space model (VSM) [109] applied to our on-line personalization context. We describe below how we separately model the profile and the query, and then how we model their combination.

Profile vector The profile of a user is modeled as a vector representing the user's tagging actions. Each element in the vector corresponds to a tag in the system, whose weight represents the number of items tagged by the user with that tag. The weight is 0 if a tag has never been used by the user. The profile of a user u_o is thus represented as follows:

$$\vec{p}(u_o) = [w_p(u_o, t_1), w_p(u_o, t_2), ..., w_p(u_o, t_{|\mathbb{T}|})],$$

where $w_p(u_o, t_n) = |\{i_m | i_m \in I(u_o), \exists \langle u_o, i_m, t_n \rangle \in \mathbb{F}\}|$. This weight $w_p(u_o, t_n)$ captures the fact that how many time the tag t_n was used by the user u_o to tag the items. The larger the weight, the more the user is interested in that tag.

Query vector A query $Q(u_o)$ issued by a querier u_o is modeled as a query vector $\vec{q}(u_o)$. Each element in the vector also corresponds to a tag in the system. The weight for each tag in the query is 1 and the weights for the other tags are 0. That is:

$$\vec{q}(u_o) = [w_q(u_o, t_1), w_q(u_o, t_2), ..., w_q(u_o, t_{|\mathbb{T}|})],$$

where $w_q(u_o, t_n) = 1$ if $t_n \in Q(u_o)$ and $w_q(u_o, t_n) = 0$ otherwise.

Hybrid interest vector The profile vector of a user is derived from a user's tagging actions and as such reflects her past and current history: it somehow represents the *sustainable* interests of the user. A query vector, on the other hand, expresses the need of a user at query time. In DT^2, we use

a *hybrid interest model*, combining the profile and the query to compute an *interest vector*. This is then used to select the most qualified users to address the query.

More specifically, we model the hybrid interest vector $\vec{l}\big(u_o, Q(u_o)\big)$ of a querier u_o at the time of her query $Q(u_o)$ as a linear combination of her profile vector $\vec{p}(u_o)$ and query vector $\vec{q}(u_o)$:

$$\vec{l}\big(u_o, Q(u_o)\big) = (1-\alpha)\frac{\vec{p}(u_o)}{\|\vec{p}(u_o)\|} + \alpha\left(\vec{w}\big(u_o, Q(u_o)\big) \cdot \frac{\vec{q}(u_o)}{\|\vec{q}(u_o)\|}\right) \tag{3.1}$$

Two complementary weighting aspects are crucial in the expression of the hybrid interest (Formula 3.1).

(1) The relative importance of tags Before combining the profile and the query, we adjust the importance of tags in the query using the Hadamard product of the query vector and the weight vector $\vec{w}\big(u_o, Q(u_o)\big)$. More specifically, each tag t_n in the query vector $\vec{q}(u_o)$ is weighted by multiplying the weight $w\big(t_n, Q(u_o)\big)$ in the corresponding position in its weight vector.

The rationale of this design choice is the following. If querier u_o already tagged several items with a given tag of the query, giving this tag a high weight does not bring much useful information in addition to the profile. In contrast, a high weight for a rarely used tag helps involving users that are interested in this tag in the query processing. We thus weight each element t_n in the query vector by its self-information $-log\mathcal{P}(t_n)$, i.e.,

$$\vec{w}\big(u_o, Q(u_o)\big) = \big[w\big(t_1, Q(u_o)\big), w\big(t_2, Q(u_o)\big), ..., w\big(t_{|\mathbb{T}|}, Q(u_o)\big)\big],$$

where $w\big(t_n, Q(u_o)\big) = -log\mathcal{P}(t_n)$ if $t_n \in Q(u_o)$ and $w\big(t_n, Q(u_o)\big) = 0$ otherwise. $\mathcal{P}(t_n)$ is the fraction of the occurrence of tag t_n in the overall behavior ($\vec{p}(u_o)$ and $\vec{q}(u_o)$) of the querier u_o, which is computed by the following formula:

$$\mathcal{P}(t_n) = \frac{1 + w_p(u_o, t_n)}{\sum_{t_m \in T(u_o)} w_p(u_o, t_m) + \sum_{t_m \in Q(u_o)} w_q(u_o, t_m)}.$$

The reason for using both the profile and the query to compute $\mathcal{P}(t_n)$ is that $\mathcal{P}(t_n)$ would be 0 for tags that are never used by u_o before but used in $Q(u_o)$, if only the profile is considered. While taking the query into account remains coherent with its meaning.

(2) The importance of the query w.r.t the profile The combination factor α ($0 \leq \alpha \leq 1$) in Formula 3.1 represents the weight given to the query with respect to the querier's profile. The larger α, the more emphasis is put on the query when selecting the relevant users to answer the query. On the other hand, $\alpha = 0$ means that the query is not taken into account in the querier's interest, which corresponds to an off-line personalization using only the querier's profile.

Given a querier u_o and a query $Q(u_o)$, we dynamically compute the combination factor α as follows:

$$\alpha = 1 - cos\big(\vec{p}(u_o), \vec{q}(u_o)\big). \tag{3.2}$$

The rationale here is the following. Basically, $cos\big(\vec{p}(u_o), \vec{q}(u_o)\big)$ measures the cosine similarity between the querier u_o's profile vector and query vector. The larger the similarity, the more correlated the query to the querier's profile, and the smaller the combination factor α will be given to the query when building the interest vector. The more correlated a query to the profile, the less additional information the query provides over the profile. Conversely, the less correlated a query to the

profile, the more additional information the tags of the query bring to select relevant users, which would not have been considered with the profile only, to process the query. In addition, dynamically adjusting the value of α according to the similarity between the profile and the query avoids the potentially training phase for a fixed α, for which it is in fact difficult to suit all the queries issued by different users.

It is important to note that both the profile vector $\vec{p}(u_o)$ and the query vector $\vec{q}(u_o)$ are normalized by dividing their norms, i.e., $\|\vec{p}(u_o)\|$ and $\|\vec{q}(u_o)\|$. The goal of this normalization is to prevent dominant tags with high weights in the profile vector to bias the overall interest. The impact of α and the weighting mechanism are measured in Section 3.2.6.2.

3.2.3 Selecting the top-k$_1$ users

3.2.3.1 Similarity among users

Our DT^2 algorithm selects the most relevant users to form the personal network and processes a query $Q(u_o)$, issued by a querier u_o, based on $\vec{l}\big(u_o, Q(u_o)\big)$ as computed above with Formula 3.1. We use the cosine similarity to compute the similarity between users. More specifically, the similarity of a user u_d with a querier u_o issuing a query $Q(u_o)$ is defined as the cosine of u_o's interest vector and u_d's profile vector[1], i.e.,

$$Similarity\big(\langle u_o, Q(u_o)\rangle, u_d\big) = cos\big(\vec{l}\big(u_o, Q(u_o)\big), \vec{p}(u_d)\big) \tag{3.3}$$

The fact that query of u_d is not taken into account in the similarity measure is deliberate: even if $\vec{l}\big(u_o, Q(u_o)\big)$ and $\vec{l}\big(u_d, Q(u_d)\big)$ are closely correlated to each other, it may only reflect the fact that they are issuing similar queries. Such users (u_d) are not necessarily helpful to answer the query $Q(u_o)$ if they have not tagged many items on the searched topics. Only the items tagged by the tags in the query are possible to appear in the query results, while such information is only reflected in the users' profiles. Moreover, the expecting items of u_o and u_d by issuing similar queries may be completely different due to the diversity of their preferences. Therefore, even if one of them finds the desired item, she may still not help the other user to get satisfactory result for her own query.

The k_1 users having the highest similarity with the pair $\langle u_o, Q(u_o)\rangle$ form the personal network used to serve the query $Q(u_o)$.

3.2.3.2 Inverted lists of users

Inspired from classical algorithms that merge inverted lists for ranking and retrieving the top-k items, we perform a top-k processing on the users of the system to select the most qualified k_1 users for answering a query using the similarity defined in Formula 3.3. The originality of our approach is to encode the tagging profiles of the users as inverted lists and apply a classical Threshold Algorithm (TA) [46] to this context. To this end, we derive from Formula 3.3 an aggregation function representing the similarity measure between users. We prove the monotonicity of this function, which is key to the correct application of TA.

In Formula 3.3, the similarity between $\langle u_o, Q(u_o)\rangle$ and u_d is defined as the cosine of u_o's interest vector and u_d's profile vector. As all the profile vectors are compared to the interest vector of u_o,

[1]The measure depends on the query and is thus not symmetric. Even if two users u_o and u_d issue queries at the same time, the similarity of u_d with u_o is different from the similarity of u_o with u_d.

$Similarity\big(\langle u_o, Q(u_o)\rangle, u_d\big)$ is a function of $\vec{p}(u_d)$. Let

$$\bar{\vec{p}}(u_d) = \frac{\vec{p}(u_d)}{\|p(u_d)\|} = [\bar{w}_p(u_d, t_0), ..., \bar{w}_p(u_d, t_{|\mathbb{T}|})],$$

$Similarity\big(\langle u_o, Q(u_o)\rangle, u_d\big)$ is also a function of $\bar{\vec{p}}(u_d)$, i.e.,

$$f\big(\bar{\vec{p}}(u_d)\big) = Similarity\big(\langle u_o, Q(u_o)\rangle, u_d\big)$$

Theorem 3.2.1. *The aggregation function $f\big(\bar{\vec{p}}(u_o)\big)$ is monotone.*

Proof. Let $L(u_o) = T(u_o) \cup Q(u_o)$,

$$\begin{aligned}
& f\big(\bar{\vec{p}}(u_o)\big) = cos\big(\vec{l}\big(u_o, Q(u_o)\big), \vec{p}(u_d)\big) \\
= \quad & \frac{\sum_{t_n \in L(u_o) \cap T(u_d)} w_l(u_o, t_n) w_p(u_d, t_n)}{\sqrt{\sum_{t_n \in L(u_o)} w_l^2(u_o, t_n)} \sqrt{\sum_{t_n \in T(u_d)} w_p^2(u_d, t_n)}} \\
= \quad & \sum_{t_n \in L(u_o) \cap T(u_d)} \frac{w_l(u_o, t_n)}{\sqrt{\sum_{t_m \in L(u_o)} w_l^2(u_o, t_m)}} \frac{w_p(u_d, t_n)}{\sqrt{\sum_{t_m \in T(u_d)} w_p^2(u_d, t_m)}} \\
= \quad & \sum_{t_n \in L(u_o) \cap T(u_d)} \bar{w}_l(u_o, t_n) \bar{w}_p(u_d, t_n),
\end{aligned}$$

where

$$\begin{aligned}
\bar{w}_l(u_o, t_n) &= \frac{w_l(u_o, t_n)}{\sqrt{\sum_{t_m \in L(u_o)} w_l^2(u_o, t_m)}}, \\
\bar{w}_p(u_d, t_n) &= \frac{w_p(u_d, t_n)}{\sqrt{\sum_{t_m \in T(u_d)} w_p^2(u_d, t_m)}}
\end{aligned}$$

In fact, $\bar{w}_l(u_o, t_n)$ corresponds to the value of tag t_n in the normalized interest vector of u_o, while $\bar{w}_p(u_d, t_n)$ is the value of tag t_n in the normalized profile vector of u_d. As $\bar{w}_l(u_o, t_n)$ can be considered as a constant for any user u_d ($d \neq o$), given any two users u_d and u_e, if for each $\bar{w}_p(u_d, t_n)$ and $\bar{w}_p(u_e, t_n)$, $t_n \in L(u_o) \cap T(u_d)$, we have $\bar{w}_p(u_d, t_n) \leq \bar{w}_p(u_e, t_n)$, then

$$\bar{w}_l(u_o, t_n) \bar{w}_p(u_d, t_n) \leq \bar{w}_l(u_o, t_n) \bar{w}_p(u_e, t_n).$$

Hence

$$\sum_{t_n \in L(u_o) \cap T(u_d)} \bar{w}_l(u_o, t_n) \bar{w}_p(u_d, t_n) \leq \sum_{t_n \in L(u_o) \cap T(u_e)} \bar{w}_l(u_o, t_n) \bar{w}_p(u_e, t_n).$$

Therefore, $f\big(\bar{\vec{p}}(u_d)\big) \leq f\big(\bar{\vec{p}}(u_e)\big)$ and $f\big(\bar{\vec{p}}(u_d)\big)$ is monotone. □

As we pointed out, the monotonicity of the aggregation function is crucial for the correctness of the threshold algorithm. We explain in the following how we build the inverted lists in such a way that this monotonicity is preserved. The inverted lists are built, one for each tag, for all the tags used by at least one user in the system. Each inverted list contains all the users who have used the corresponding tag. The score of each user u_d in an inverted list is the value of the tag in her

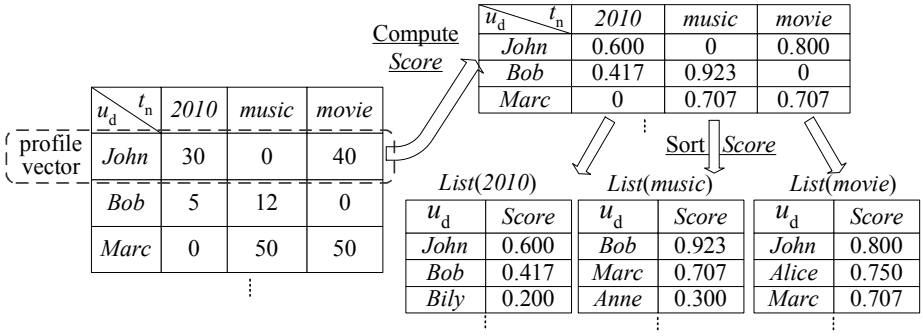
normalized profile vector $\bar{\vec{p}}(u_d)$. The users are sorted in descending order of their scores. More formally, for each $t_n \in \mathbb{T}$,

$$List(t_n) = \{\langle u_1, score_{t_n}(u_1)\rangle, ..., \langle u_d, score_{t_n}(u_d)\rangle, ...\},$$

where $u_d \in \{u_d | u_d \in \mathbb{U}, \exists \langle u_d, i_m, t_n\rangle \in \mathbb{F}\}$, $score_{t_n}(u_d) = \bar{w}_p(u_d, t_n)$ and $score_{t_n}(u_d) \geq score_{t_n}(u_{d+1})$.

Going back to the proof of Theorem 3.2.1, we observe that if a user u_d appears after a user u_e in all the inverted lists, i.e., $\bar{w}_p(u_d, t_n) \leq \bar{w}_p(u_e, t_n)$ for all t_n, u_d would not be ranked higher than u_e when selecting the top k_1 users according to their similarity with $\langle u_o, Q(u_o)\rangle$. This allows the threshold algorithm to prune the unqualified users earlier. Note that the score of user u_d in the list $List(t_n)$ is assigned as the value of t_n in u_d's normalized profile vector $\bar{w}_p(u_d, t_n)$ rather than the corresponding value in her profile vector $w_p(u_d, t_n)$. This is due to the fact that $w_p(u_d, t_n) \leq w_p(u_e, t_n)$ does not necessarily imply $\bar{w}_p(u_d, t_n) \leq \bar{w}_p(u_e, t_n)$ and may thus jeopardize the monotonicity of the similarity measure.

Figure 3.2 gives a simple example of how the inverted lists are derived for the users' profile vectors. Taking *John* for example, we first compute the score for each tag in her profile vector by dividing the number of tagging actions related to that tag by the norm of her profile vector and obtain: $\bar{w}_p(John, 2010) = 30/50 = 0.6$, $\bar{w}_p(John, music) = 0/50 = 0$ and $\bar{w}_p(John, movie) = 40/50 = 0.8$. After computing the scores for all the users in the system, these are organized in per tag list, containing all the users with positive value for that tag, and then ranked in their descending order to form the final inverted lists.



u_d \ t_n	2010	music	movie
John	30	0	40
Bob	5	12	0
Marc	0	50	50

u_d	2010	music	movie
John	0.600	0	0.800
Bob	0.417	0.923	0
Marc	0	0.707	0.707

u_d	Score
John	0.600
Bob	0.417
Bily	0.200

u_d	Score
Bob	0.923
Marc	0.707
Anne	0.300

u_d	Score
John	0.800
Alice	0.750
Marc	0.707

Figure 3.2: Example of inverted list computation

Note that the inverted lists are common to all users, and hence do not require much storage: at most $|\mathbb{U}| \times |\mathbb{T}|$ entries should be maintained. This is in fact a very loose upper bound as the use of tags follows a long-tail distribution.

3.2.3.3 Ranking users

When a user u_o issues a query, DT^2 first builds a hybrid interest vector using her profile and the query as described in Formula 3.1. Then DT^2 finds the top k_1 users for answering the query. This is achieved by first identifying the inverted lists of the tags appearing in u_o's hybrid interest vector. The scores for tags that do not appear in the hybrid interest vector do clearly not contribute to the overall similarity. The lists are scanned in a breadth-first manner, i.e., the first entries in each list are

scanned first, and then the second entries are scanned and so on. The querier is skipped when she is encountered in the inverted lists during scanning. The term *sequential access* is used to refer to the read operation of a user in the inverted lists.

A heap of k_1 entries in the form of pair $\langle u_d, Similarity(\langle u_o, Q(u_o)\rangle, u_d)\rangle$ is maintained and these entries are sorted in descending order of $Similarity(\langle u_o, Q(u_o)\rangle, u_d)$. Once a user u_d different from u_o is seen in a list under sequential access, if she does not exist in the heap, her similarity with $\langle u_o, Q(u_o)\rangle$ is computed using Formula 3.3. We use the term *random access* to refer to the call of the function using Formula 3.3 to compute the similarity. If the obtained similarity of u_d is larger than that of the k_1th user, the k_1th entry in the heap is replaced by the new entry. Otherwise, the new entry is discarded.

The algorithm stops as soon as the k_1th user's similarity is not smaller than a threshold. The threshold is obtained by computing the cosine similarity between u_o's hybrid interest vector and a virtual profile vector where the values for the tags related to the inverted lists under scanning are the last seen values in each of the lists and those for other tags are 0, i.e.,

$$threshold = \sum_{t_n \in T(u_o) \cup Q(u_o)} \bar{w}_l(u_o, t_n) b(t_n),$$

where $b(t_n)$ is the last seen value in $List(t_n)$. The monotonicity of the similarity measure ensures that users appearing after $b(t_n)$ in all $List(t_n)$ ($t_n \in T(u_o) \cup Q(u_o)$) would not have a higher similarity than the threshold.

For the tags used by few users, the corresponding inverted lists may be very short. Once the cursor reaches the end of those lists during scanning, the last seen values will no longer change. This would keep the threshold high, and incur unnecessary computation as the users that did not use these tags at all are still supposed to have the same level of interest in these tags, as indicated by the last seen values. Therefore, $b(t_n)$ is set to 0 when the end of $List(t_n)$ is reached. This does not affect the correctness of the top k_1 selected users as the corresponding values in the profile vectors of the users who do not appear in these lists are exactly 0. Algorithm 3.1 presents the user selection procedure. We use $Network(u_o, Q(u_o))$ to denote the personal network containing the k_1 users to process the user u_o's query $Q(u_o)$.

Algorithm 3.1 mimics the threshold algorithm in [46] and is in the same sense *instance optimal* (i.e., modulo a constant number of sequential accesses). In fact, several optimizations are possible to incorporate into the basic algorithm. For instance, we do not need to rank the candidate heap each time a user is seen but only after a bunch of users are discovered. These optimizations are potentially parts of the future work.

3.2.4 Selecting the top-k_2 items

3.2.4.1 Ranking items

The last phase of DT^2 consists in finding the k_2 items that are the most relevant to the query. The search space is now reduced to the profiles of the selected k_1 users only.

We adapt a widely used scoring function [110] to rank the items. More specifically, the score of an item i_m for querier u_o's query $Q(u_o)$ is computed as the sum of the scores given to i_m by each user u_d in u_o's personal network, which is the number of tags used by u_d to tag i_m times the

Algorithm 3.1 Top k_1 user selection

1. **Input:** u_o's query $Q(u_o)$
2. **Output:** $Network\big(u_o, Q(u_o)\big)$
3. $ScanLists \leftarrow \{List(t_n) | \forall t_n \in T(u_o) \cup Q(u_o)\}$
4. $ScanPosition \leftarrow 1$
5. **loop**
6. **for** each $List(t_n)$ in $ScanLists$ **do**
7. get user u_d in $ScanPosition$ of $List(t_n)$
8. **if** u_d exists and $u_d \neq u_o$ **then**
9. $threshold \leftarrow threshold - \bar{w}_l(u_o, t_n)\big(b(t_n) - score_{t_n}(u_d)\big)$
10. $b(t_n) \leftarrow score_{t_n}(u_d)$
11. **if** u_d not in the heap **then**
12. compute $Similarity\big(\langle u_o, Q(u_o)\rangle, u_d\big)$
13. **if** $Similarity\big(\langle u_o, Q(u_o)\rangle, u_d\big) > Similarity\big(\langle u_o, Q(u_o)\rangle, u_{k_1}\big)$ **then**
14. replace u_{k_1} by u_d and sort the heap
15. **end if**
16. **end if**
17. **else if** u_d does not exist **then**
18. $b(t_n) \leftarrow 0$
19. **end if**
20. **if** $Similarity\big(\langle u_o, Q(u_o)\rangle, u_{k_1}\big) \geq threshold$ **then**
21. return the k_1 users in the heap as $Network\big(u_o, Q(u_o)\big)$
22. **end if**
23. **end for**
24. $ScanPosition \leftarrow ScanPosition$+1
25. **end loop**

similarity between $\langle u_o, Q(u_o)\rangle$ and u_d, i.e.,

$$Score\big(Q(u_o), i_m\big) = \sum_{u_d \in Network(u_o, Q(u_o))} Score\big(u_d, Q(u_o), i_m\big) Similarity\big(\langle u_o, Q(u_o)\rangle, u_d\big), \quad (3.4)$$

where $Score\big(u_d, Q(u_o), i_m\big) = |\{\langle u_d, i_m, t_n\rangle | t_n \in Q(u_o)\}|$.

Since the top-k_2 query is processed using a network of k_1 users selected on-line, there is no need to pre-compute per tag inverted lists as the traditional approaches to enable efficient top-k processing. The mere reading of tagging actions is enough to gradually compute the relevance score of each candidate item and find the top k_2 relevant items. As we will see later, computing the inverted lists requires in fact more computation. Moreover, since the personal networks computed on-line are query dependent, the probability that consecutive queries of the same user have exactly the same personal network is very low. Pre-computed inverted lists would then hardly be re-used and thus unworthy.

Instead, the relevance score of each item in DT^2 is obtained by directly iterating over the tagging actions of each user in the personal network, which is specifically tailored for a pair $\langle user, query\rangle$. Once the relevance scores of all candidate items are computed, the k_2 items with the highest scores are selected by partially sorting these items. Note that since only a small size of the personal network is enough to achieve good results, this operation is reasonable. Algorithm 3.2 depicts the top k_2 item selection procedure.

Algorithm 3.2 Top k_2 item selection

1. **Input:** $Network\big(u_o, Q(u_o)\big)$ and $Q(u_o)$
2. **Output:** top k_2 items
3. **for** each u_d in $Network\big(u_o, Q(u_o)\big)$ **do**
4. $F(u_d, Q(u_o)) \leftarrow$ get $\langle u_d, i_m, t_n \rangle$ for all $t_n \in Q(u_o)$
5. **for** each i_m in $F(u_d, Q(u_o))$ **do**
6. compute $Score\big(u_d, Q(u_o), i_m\big)$
7. update $Score\big(Q(u_o), i_m\big)$ in the heap
8. **end for**
9. **end for**
10. **for** each i_m ($m > k_2$) in the heap **do**
11. **if** $Score\big(Q(u_o), i_m\big) > Score\big(Q(u_o), i_{k_2}\big)$ **then**
12. replace the k_2th item i_{k_2} in the heap by i_m
13. sort the first k_2 items in the heap
14. **end if**
15. **end for**
16. return the first k_2 items in the heap

3.2.4.2 Cost analysis

As described above, finding the top k_2 items consists in iterating over the tagging actions of the k_1 selected users, computing the relevance scores of the items appearing in their profiles and ranking them accordingly. Supposing that for a user u_o's query $Q(u_o)$, each of the k_1 selected users has $N_t(Q(u_o))$ tagging actions with a tag in $Q(u_o)$, and the cost for reading a tagging action and updating the relevant score of the corresponding item is c_c, then the cost for computing the score of all the candidate items can be described as

$$C_c(Q(u_o)) = c_c \times k_1 \times N_t(Q(u_o)). \tag{3.5}$$

If $N_i(Q(u_o))$ items are involved in these tagging actions, $N_i(Q(u_o))$ entries, containing the identifier and the score of each item, should be maintained in the heap. In the worst case, when the score of each item out of the first k_2 items in the heap is larger than that of the k_2th item, $N_i(Q(u_o)) - k_2$ sorting operations are necessary to rank these items. Knowing that a single sorting over a sorted list of k_2 items can be achieved in $O(logk_2)$, assuming the cost of each sorting operation is c_s, the cost of ranking the top k_2 items can be written as

$$C_s(Q(u_o)) = \big(N_i(Q(u_o)) - k_2\big) \times c_s \times logk_2. \tag{3.6}$$

Therefore, given a personal network of k_1 users, the total cost for obtaining the top k_2 items is the sum of $C_c(Q(u_o))$ and $C_s(Q(u_o))$.

If the inverted lists are computed before processing the query, assuming that each item i_m is tagged by $N_t(i_m)$ tags in the query $Q(u_o)$, the storage for the associated lists would be $N_t(i_m)$ times of the heap size in DT^2, as each item appears in $N_t(i_m)$ lists where it is tagged. Assuming that all the inverted lists have the same length, i.e., $\frac{N_i(Q(u_o)) \times N_t(i_m)}{|Q(u_o)|}$, the cost for sorting one list $L(t_n)$ would be

$$C_s(L(t_n)) = c_s \times \frac{N_i(Q(u_o)) \times N_t(i_m)}{|Q(u_o)|} log\big(\frac{N_i(Q(u_o)) \times N_t(i_m)}{|Q(u_o)|}\big).$$

Thus the cost for sorting all the query related lists can be written as

$$C_s(L(Q(u_o))) = |Q(u_o)| \times c_s \times \frac{N_i(Q(u_o)) \times N_t(i_m)}{|Q(u_o)|} log\big(\frac{N_i(Q(u_o)) \times N_t(i_m)}{|Q(u_o)|}\big).$$

With $k_2 << N_i(Q(u_o))$ and $N_t(i_m) \approx |Q(u_o)|$, $C_s(L(Q(u_o)))$ is about $\frac{N_t(i_m) log N_i(Q(u_o))}{log k_2}$ times more than $C_s(Q(u_o))$ in DT^2. This does not account for the cost of processing the query over these lists. Considering that the cost for reading the tagging actions and computing the relevance scores of the items is similar in both cases, this fully justifies our choice for the item top-k algorithm.

3.2.5 Adjusting the personal network

The key idea of DT^2 lies in performing the personalization on-line by first associating the querier with a personal network of the top k_1 users and then processing the query within this network to find the top k_2 items.

3.2.5.1 Design choices

Unlike other top-k algorithms that also aim to obtain the k_2 most relevant items, DT^2 restricts the search space only to the personal network of the top k_1 users. Thus, selecting an appropriate value for k_1 is crucial to the result quality: *(i)* too small value of k_1 would exclude some relevant items out of the candidate set so that they could not appear in the results regardless of the value of k_2; *(ii)* too large value of k_1 may introduce unnecessary noise to the candidate set by taking into account the opinions of users with disjoint or even opposite interests and thus dilute the positive effect of personalization. In addition, users behave differently in different applications and it is important to adjust the value of k_1 for each user according to her current situation. Furthermore, the larger k_1, the longer it requires to find these users and derive the top k_2 items from their tagging profiles. It is thus important to achieve a balance between the result quality and the processing time. We discuss in the following how the parameter k_1 can be adaptively adjusted so as to maximize the system performance and the user satisfaction.

3.2.5.2 Incremental adjustment

In DT^2, the users are folded into a querier's personal network in an incremental manner. More users are used for finding the top k_2 items only if the querier appears to be not satisfied with the current results obtained with a smaller personal network. Figure 3.3 illustrates how the size of personal network is iteratively adjusted in DT^2. We use directly, in the figure, the hybrid interest vector as an input for the ease of presentation.

The system begins to process a querier's query by selecting a small number of users, noted as $k_1^{(0)}$, and finding the top k_2 items with the profiles of the $k_1^{(0)}$ users. The value of $k_1^{(0)}$ can be empirically obtained through the statistics of query histories of either individual user or all users in the system. If the querier is not satisfied with the top k_2 items, the system continues to select $k_1^{(1)} - k_1^{(0)}$ ($k_1^{(1)} > k_1^{(0)}$) additional users in the next round and show the new top k_2 items derived from all the $k_1^{(1)}$ users' profiles to the querier. This process continues until the user is satisfied or all candidate users have already been exhausted and considered in the querier's personal network: after the rth round, the number of users having $Similarity\big(\langle u_o, Q(u_o)\rangle, u_d\big) > 0$ is smaller than $k_1^{(r)}$.

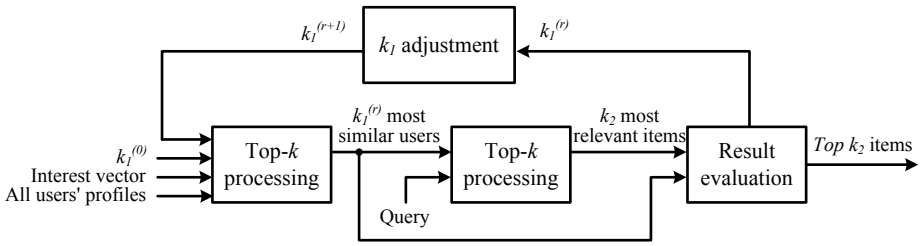


Figure 3.3: Adjusting the personal network (k_1)

Both the computation of the additional users and that of the new items could actually be achieved in an incremental manner. However, to continue the user selection without missing any qualified user, all users that have been explored in the previous sequential access phase (previous round) are maintained, i.e., we do not only maintain the k_1 users with the highest similarities when a fixed k_1 is used like in Algorithm 3.1.

Algorithm 3.3 depicts how the additional users are gradually involved in the querier's personal network in the $(r+1)$th ($r \geq 0$) round. We invoke Algorithm 3.1 in Algorithm 3.3 for the ease of presentation, but the heap size is not limited to $k_1^{(r)}$ at the rth round. Indeed, all explored users are maintained in the heap but only the first $k_1^{(r)}$ are sorted.

Algorithm 3.3 Top $k_1^{(r+1)}$ user selection

1. **Input:** user heap after the rth round and $k_1^{(r+1)}$
2. **Output:** $Network\big(u_o, Q(u_o)\big)$ with $k_1^{(r+1)}$ users
3. **if** heap size $< k_1^{(r+1)}$ **then**
4. continue the processing with Algorithm 3.1 for top $k_1^{(r+1)}$ users from last scanned positions of the rth round
5. **else**
6. sort the heap by descending $Similarity\big(\langle u_o, Q(u_o)\rangle, u_d\big)$
7. **if** $Similarity\big(\langle u_o, Q(u_o)\rangle, u_{k_1^{(r+1)}}\big) \geq threshold$ **then**
8. return top $k_1^{(r+1)}$ users in heap as $Network\big(u_o, Q(u_o)\big)$
9. **else**
10. continue the processing with Algorithm 3.1 for top $k_1^{(r+1)}$ users from last scanned positions of the rth round
11. **end if**
12. **end if**

To merge the new users' profiles of a given round with the results of the previous round, we follow the principle of Algorithm 3.2: for each item appearing in the additional profiles, if it is already in the item heap after the rth round, its relevance score is updated by taking the new information into account; otherwise, it is added to the heap using Formula 3.4. Once all the scores are completely computed, the new top k_2 items are obtained by partially sorting the heap.

The latency of processing the query first on a personal network of $k_1^{(r)}$ users and then on a personal network of $k_1^{(r+1)}$ users mainly depends on the time to sort the first k_2 items in the personal network of $k_1^{(r)}$ users. There is almost no difference between finding directly the top $k_1^{(r+1)}$ users and finding

them incrementally. Because continuing the processing only requires to sort the users in the heap once if necessary (Algorithm 3.3, line 5-12), otherwise exactly the same processing is carried out to obtain the top $k_1^{(r+1)}$ users. The only exception is that the top $k_1^{(r+1)}$ users can be selected from the heap without further processing (line 7-8). In this case, the similarities of certain users are computed in vain, which also requires longer time for sorting. However, the probability that the top-k users are easier to find than the top-k' users with $k > k'$ should be low. Denoting by $N_i^{(r)}(Q(u_o))$ the size of the item heap at the rth round, this latency can be expressed as Formula 3.6, with $N_i(Q(u_o)) = N_i^{(r)}(Q(u_o))$. If a querier's query is answered in R round, the total latency comparing to processing it directly on a personal network of $k_1^{(R)}$ users is

$$Latency(Q(u_o)) = \sum_{r=0}^{R-1} \left(N_i^{(r)}(Q(n_o)) - k_2\right) \times c_s \times log k_2.$$

3.2.6 Experimental evaluation

We evaluate DT^2 in the context of collaborative tagging systems and compare it with tag-based non-personalized and off-line personalized approaches.

3.2.6.1 Experimental setup

Datasets and query generation. We evaluate DT^2 using real traces from *CiteULike* and *delicious*, both being collaborative tagging sites through which users bookmark, annotate and share, respectively, research papers and URLs.

The *CiteULike* trace involves $33,834$ users and was downloaded directly from the site in October 2008. The *delicious* trace involves $137,897$ users and was crawled in January 2009. We randomly selected $10,000$ resp. $50,000$ users who participated in at least 10 tagging actions to run the simulations. The figures of the datasets are depicted in Table 3.1.

Table 3.1: Datasets

Data	Users	Items	Tags	Tagging actions
CiteULike	$10,000$	$852,581$	$189,378$	$3,084,050$
delicious	$50,000$	$3,103,204$	$687,458$	$24,626,054$

Selecting a good set of top-k queries and results is not an easy task, especially when the results should meet the need of individual users who may exhibit different preferences facing a same query. To address this issue, we generate queries *for each user* as follows: an item tagged by at least one other user with the same tag is picked randomly from the items tagged by this user; the query is generated with the tags used by this user to annotate this item; the item and all related tagging actions of this user are removed from her profile and are excluded when computing her profile vector. The underlying intuition is that the tags used by a user to annotate an item are often those she feels appropriate to describe that item. Thus, they are likely to be the words she previously used to search for that item. Figure 3.4 depicts the correlation between the generated queries and the profiles. The bar at $S = x$ corresponds to the percentage of queries whose similarities with the querier's profile are within the interval $(x - 0.2, x]$. Note that the bar at $S = 0$ corresponds to the interval $(-0.2, 0]$. Yet, as the cosine similarity cannot be negative, the queries fall in this interval is

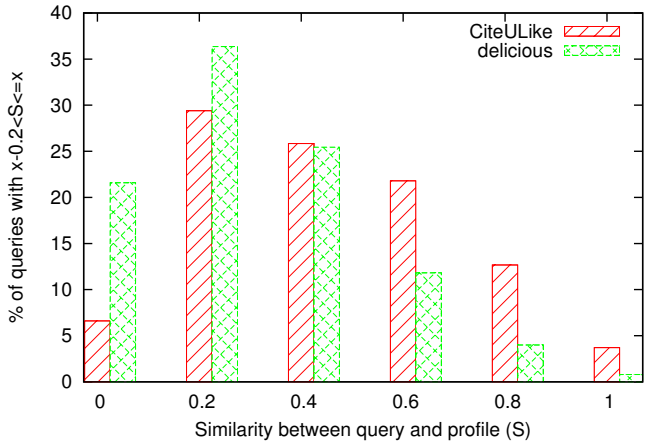


Figure 3.4: Distribution of query similarity

equivalent to those having similarity 0 with the querier's profile. We use $S = 0$ in the following to avoid the misunderstanding.

Evaluation methodology We compare DT^2 against both non-personalized and off-line personalized approaches (tag-based) along the following properties: *(i)* result quality, *(ii)* storage and *(iii)* processing time.

Result quality. We evaluate the result quality achieved by DT^2 as its ability to *retrieve* and *highly rank* the item initially removed from the querier's profile. To this end, we use both the average recall [107] and the mean reciprocal rank (MRR) [111] metrics to quantify it.

MRR has been used for applications where there is typically a single relevant item. The reciprocal rank of a querier's query is the inverse of the rank of the removed item in the result list. If the relevant item does not appear in the result list, the reciprocal rank is counted as 0. The mean reciprocal rank is the average of the reciprocal ranks over all queries. The larger the MRR, the higher the desired items are ranked.

Considering that users tend to look at only the top part of the ranked result list to find relevant items, we also use the average recall in the top-k items over all queries as a measure of the result quality, denoted as R_k. Recall is defined as the number of retrieved relevant items divided by the total number of existing relevant items. In our case, if the item, initially removed from the querier's profile, is in the top-k items, the recall for this query is 1. It is 0 otherwise. Note that we do not use the $precison@k$ metric to measure the result quality: considering only one relevant item for each query, the corresponding precision in the top-k items would simply be R_k/k.

Storage The inverted lists are encoded as $\langle tag, user, score \rangle$ entries in DT^2 and $\langle tag, item, score \rangle$ entries in the non-personalized and off-line personalized approaches. We measure the storage, like in [12], as the total length of inverted lists.

Processing time The time required for answering a query in DT^2 consists of *(i)* the time required to select the k_1 most similar users and *(ii)* the time required to select the k_2 most relevant items over those k_1 users' profiles. We use the wall-clock time to compare this processing time with the time to process the query in the alternative approaches. In addition, to convey the efficiency of DT^2 for selecting the top k_1 users, we also compare it against the approach in [112]. To this end, we measure the number of users for whom the similarities are computed before finding the top k_1 ones.

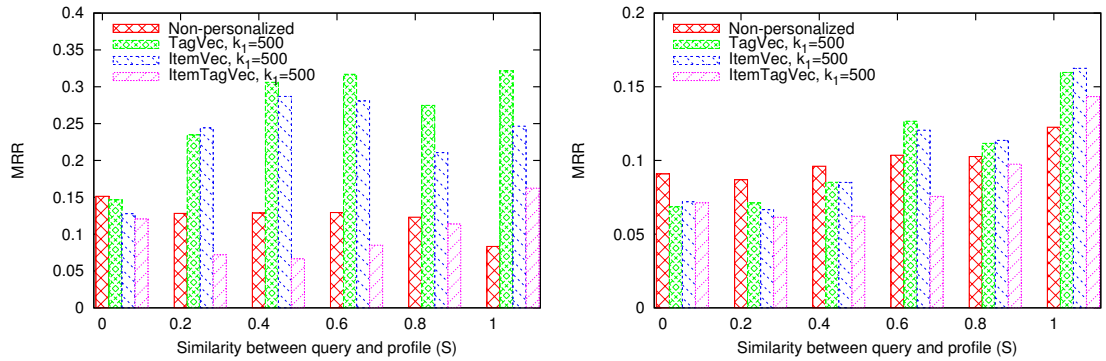
3.2.6.2 Result quality

DT^2 personalization We evaluate the effectiveness of DT^2 by comparing MRR and R_k with 3 natural candidate strategies to achieve off-line personalization: they all rely solely on tagging profiles to restrict the search space. All three strategies model user profiles as vectors, and use the cosine similarity between the vectors to determine the personal network. In the first strategy [63], denoted by *TagVec*, the same profile model as DT^2 is used, i.e., each element in the vector corresponds to a tag in the system and its weight is the number of items tagged by the user with this tag. In the second strategy [67, 12], denoted by *ItemVec*, each element in the vector corresponds to an item in the system and its weight is 1 if the user has tagged this item. The similarity between two users depends on the number of items they tagged in common. In the third strategy [67], denoted by *ItemTagVec*, each element represents a pair $\langle item, tag \rangle$ and its weight is 1 if the user has tagged this item with this tag. The similarity between two users depends on the number of times they tagged the same items with the same tag and it is the same as the one used in P4Q (Section 2.4.1).

Before proceeding, we first examine the ability of these off-line personalization strategies to deal with ambiguous queries compared to non-personalized processing. For the sake of comparison with DT^2, in both non- and off-line personalized approaches, the score of each item in the inverted lists is computed as the number of times the item has been tagged with this corresponding tag. The scoring function to rank the items is the same as the one used in DT^2. In the non-personalized approach, $Similarity\big(\langle u_o, Q(u_d)\rangle, u_d\big)$ is set to 1 to ignore the effect of personalization.

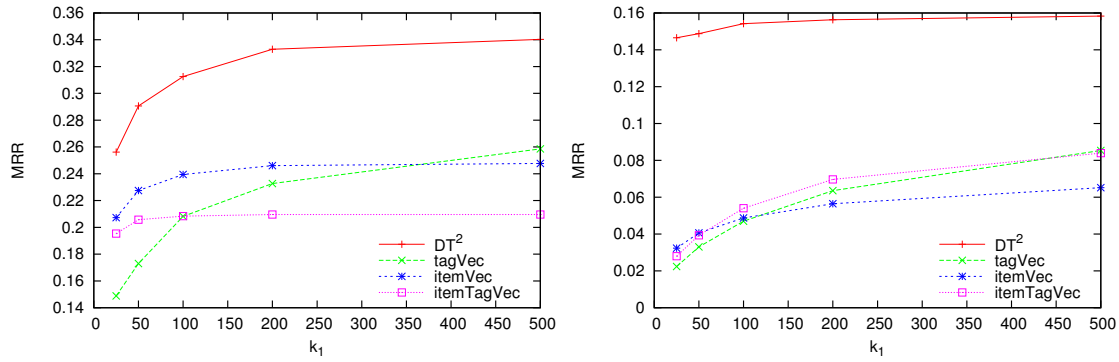
Figure 3.5 compares the MRR in the non-personalized approach and the three off-line personalized ones, with personal networks of 500 users. The queries are grouped according to their similarities with the queriers' profiles. We observe from the figure that regardless of the similarity between the query and the querier's profile, the non-personalized approach provides similar ranks for different query groups. This is consistent with the random nature of the selected queries. In contrast, the effectiveness of the three off-line personalized approaches highly depends on the similarity between the query and the querier's profile. If the query is correlated to the querier's profile, the off-line personalized approaches achieve up to 2.8 times better MRR for *CiteULike* and 32% better MRR for *delicious* than the non-personalized approach. This is due to the fact that the off-line personalized approaches do not consider the irrelevant information introduced by the users having few common interests with the querier. However, for queries that are not correlated to the querier's profile, typically corresponding to the emerging interests, the off-line personalization degrades the MRR up to 48% for *CiteULike* and 37% for *delicious*. Indeed, the search space is restricted to similar users while the relevant items are likely to belong to users whose profile is similar to the query. This observation precisely motivates our on-line personalization approach.

When comparing DT^2 to the off-line personalized approaches, we observe from Figure 3.6 that the more users are selected to answer the query, the better the MRR. Moreover, regardless of



(a) *CiteULike* (b) *delicious*

Figure 3.5: Non- vs. off-line personalized approaches

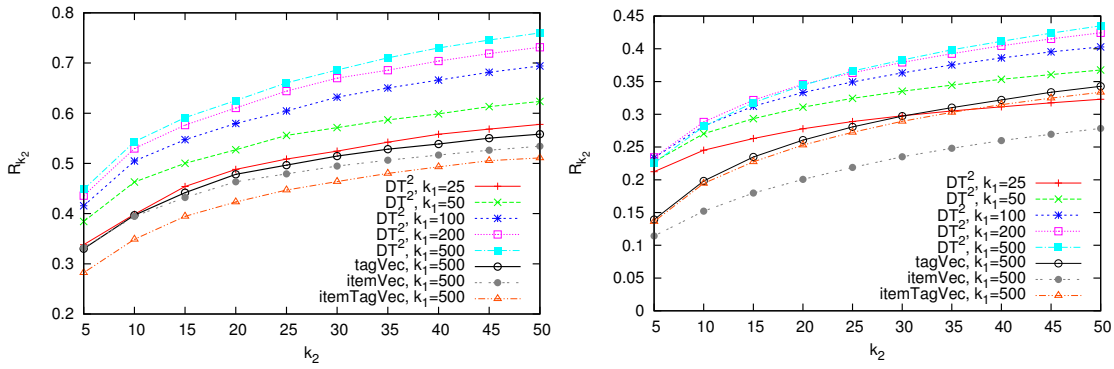


(a) *CiteULike* (b) *delicious*

Figure 3.6: Off-line personalized approaches vs. DT^2 in MRR

this number (k_1), DT^2 clearly outperforms the three off-line personalized strategies: the MRR is consistently more than 30% better than any other off-line personalized approach in both traces. The fewer users are selected to answer a query, the more significant the difference. This is due to the fact that when the query result only relies on a small number of users, the relative importance of each user is higher. With only 25 users in DT^2, the MRR is even better than in the off-line personalized approaches involving 500 users. Clearly, considering both the query and the profile while modeling the interest and selecting the users accordingly provides more qualified users to answer the queries.

The MRR conveys the ability of DT^2 to retrieve and rank the relevant item of each query high. Figure 3.7 further show how DT^2 allows to obtain better results if only the top k_2 items are returned. Not surprisingly, regardless of the personalization strategy, the more items are returned to the user, the better the average recall. Moreover, with the same number of k_2, the average R_{k_2} obtained in DT^2 with only 25 users is always better than that in the three off-line personalized approaches with



(a) *CiteULike* (b) *delicious*

Figure 3.7: Off-line personalized approaches vs. DT^2 in $recall$

500 users. Better results can be observed if more users are selected for the query processing in DT^2. For the ease of presentation, we use R_{10} instead of the recall with various k_2 in the following.

When focusing on the queries that are not correlated to the querier's tagging history (e.g. emerging interests), we observe in Table 3.2 and Table 3.3 that MRR and R_{10} in the personal network of 25 users in DT^2 is up to 70% (resp. 225%) and 54% (resp. 150%) better than those in the personal network of 500 users in $TagVec$, which is the best among the off-line personalized approaches, on the *CiteULike* (resp. *delicious*) trace. They are also up to 64% (resp. 146%) and 27% (resp. 46%) better than the non-personalized approach. Those results confirm the effectiveness of DT^2 to account for the emerging interests, representing 35% of the queries in *CiteULike* and 58% in *delicious* (Figure 3.4).

Note that the overall performance on the *CiteULike* trace is better than that on the *delicious* trace because the *delicious* trace is 5 times larger in terms of users than the *CiteULike* one while the

Table 3.2: MRR for queries not correlated to the profile

Similarity (S)	*CiteULike*			*delicious*		
	DT^2 $k_1 = 25$	TagVec $k_1 = 500$	Non personalized	DT^2 $k_1 = 25$	TagVec $k_1 = 500$	Non personalized
$S = 0$	0.248	0.146	0.151	0.224	0.069	0.091
$0 < S \leq 0.2$	0.252	0.234	0.128	0.132	0.062	0.087

Table 3.3: R_{10} for queries not correlated to the profile

Similarity (S)	*CiteULike*			*delicious*		
	DT^2 $k_1 = 25$	TagVec $k_1 = 500$	Non personalized	DT^2 $k_1 = 25$	TagVec $k_1 = 500$	Non personalized
$S = 0$	0.389	0.253	0.342	0.322	0.129	0.221
$0 < S \leq 0.2$	0.376	0.357	0.296	0.197	0.120	0.172

queries are generated with the same procedure. Moreover, the *delicious* data is sparser than the *CiteULike* one. Users in *delicious* are more versatile and tag various URLs while interests are most focused in *CiteULike*.

Relevance of the interest model The weights of the query while modeling the interest of a pair $\langle user, query \rangle$ is captured by the combination factor α and the weight vector $\vec{w}\big(u_o, Q(u_o)\big)$. To emphasize the impact of the dynamic adaptation of α and $\vec{w}\big(u_o, Q(u_o)\big)$ on the result quality, we evaluate the impact of α, applied to all queries in 25-user personal networks, regardless of their similarities with the querier's profile. The results are shown in Table 3.4 and Table 3.5 for MRR and R_{10} respectively. They are also compared with the case where the query tags are equally weighted, i.e., $\vec{w}\big(u_o, Q(u_o)\big) = [1, ..., 1]$. $\alpha = 0$ means that only the profile is used to select the users to process the query and $\alpha = 1$ corresponds to only considering the query. These results confirm that *(i)* considering both the query and the profile achieves better result quality than considering only either one of them; *(ii)* adjusting α depending on the similarity between the query and the profile consistently achieves the best results; *(iii)* weighting the query tags according to their occurrence in the querier's overall behavior provides better results than considering them equally.

Table 3.4: MRR for varying α

With *adaptive* tag weights in $\vec{w}\big(u_o, Q(u_o)\big)$							
α	0	0.2	0.4	0.6	0.8	1	1-S
CiteULike	0.149	0.188	0.233	0.249	0.251	0.246	**0.256**
delicious	0.032	0.049	0.094	0.137	0.146	0.142	0.147
With *equal* tag weights in $\vec{w}\big(u_o, Q(u_o)\big)$							
α	0	0.2	0.4	0.6	0.8	1	1-S
CiteULike	0.149	0.184	0.229	0.244	0.251	0.241	0.251
delicious	0.032	0.048	0.090	0.129	0.139	0.140	0.136

Table 3.5: R_{10} for varying α

With *adaptive* tag weights in $\vec{w}\big(u_o, Q(u_o)\big)$							
α	0	0.2	0.4	0.6	0.8	1	1-S
CiteULike	0.249	0.294	0.360	0.385	0.391	0.384	**0.397**
delicious	0.066	0.096	0.169	0.234	0.244	0.241	0.244
With *equal* tag weights in $\vec{w}\big(u_o, Q(u_o)\big)$							
α	0	0.2	0.4	0.6	0.8	1	1-S
CiteULike	0.249	0.294	0.351	0.362	0.372	0.372	0.384
delicious	0.066	0.087	0.149	0.203	0.214	0.213	0.209

3.2.6.3 Storage and processing time

We evaluate the storage requirement and the query processing time of DT^2 against *(i)* non-personalized approach and *(ii)* the *TagVec* off-line personalized approach, which returns the best results among the off-line personalized approaches.

Storage requirement DT^2 makes use of inverted lists of users, one for each tag. On the *CiteULike* trace, $189,378$ inverted lists are maintained and the total length of these lists is $589,740$ entries. On the *delicious* trace, $687,458$ inverted lists are maintained and the total length is $6,570,566$ entries. Note that this storage is independent of the number of users selected to answer a query, namely k_1.

In the non-personalized approach, the same number of inverted lists is maintained but the total length is $3,030,551$ entries for *CiteULike* and $16,521,714$ entries for *delicious*, which is respectively 5 and 2.5 times the storage used by DT^2. This is due to the fact that the number of items in the system is much larger than the number of users.

In *TagVec*, the storage highly depends on the size of the personal network and the way the inverted lists are built. The more users in the personal network, the more storage is needed. Here, we compare DT^2 with the most storage consuming but time efficient approach in [12] to build the inverted lists in *TagVec*: for each user, the inverted lists are built for every tag used by at least one user in her personal network. Taking 500-user personal networks on the *CiteULike* trace for example, the total length of the stored inverted lists is $1,894,929,966$ entries, which is more than 3200 times of that in DT^2. While for *delicious*, with personal networks of 500 users, the total length of the inverted lists is $24,453,501,395$ entries, which is more than 3700 times of that in DT^2. In fact, even the most storage effective approach in [12] requires the same number of entries as the non-personalized approach while each entry should contain more information in addition to the score for the personalization need. Moreover, it dramatically increases the query processing time and is thus not used as our baseline.

Performance of top-k_1 user selection We first count the number of users whose similarity should be computed to measure the efficiency of top k_1 user selection. Obviously, if no inverted lists of users are pre-computed, the system needs to traverse all the users to identify the top k_1 users. In [112], the authors proposed to only compute the similarity for users who have tagged the queried resource to generate the tag recommendation: this dramatically decreases the amount of computation. In the following, we report on our comparison of DT^2 with that in [112] by considering all the users having at least one common tag with the querier's hybrid interest vector.

Figure 3.8 shows the relation between the size of the personal network (k_1) and the number of sequential and random accesses on both the *CiteULike* and the *delicious* traces. The number of sequential accesses is only slightly more than the number of random accesses. Considering that a random access is more expensive than a sequential one [12], and a random access corresponds to a computation of similarity between users, it is reasonable to compare the efficiency of top k_1 user selection with the number of such computations.

The number of similarity computations between users increases almost linearly with the size of the personal network on *CiteULike*, but it increases slower on *delicious*. Even if 500 users should be selected, only the similarity with 6.39% and 3.68% users in the system are computed on the *CiteULike* trace and the *delicious* trace respectively. Computing the similarity with only 2.02% and 1.81% of the users is enough to select the top 25 users. With the method in [112], regardless of the size of the personal network, on average, the similarities of 42.35% and 73.07% of users in the system should be computed on *CiteULike* and *delicious* respectively. DT^2 largely outperforms this method while selecting the users to form the personal network.

Processing time The overall time needed to answer a query with DT^2 consists of the time required to select the top k_1 users and the time required to retrieve the top k_2 items. In the non-personalized

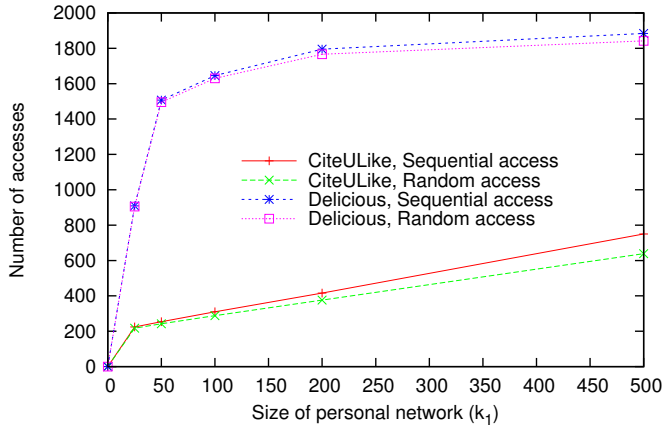


Figure 3.8: Performance of top k_1 user selection

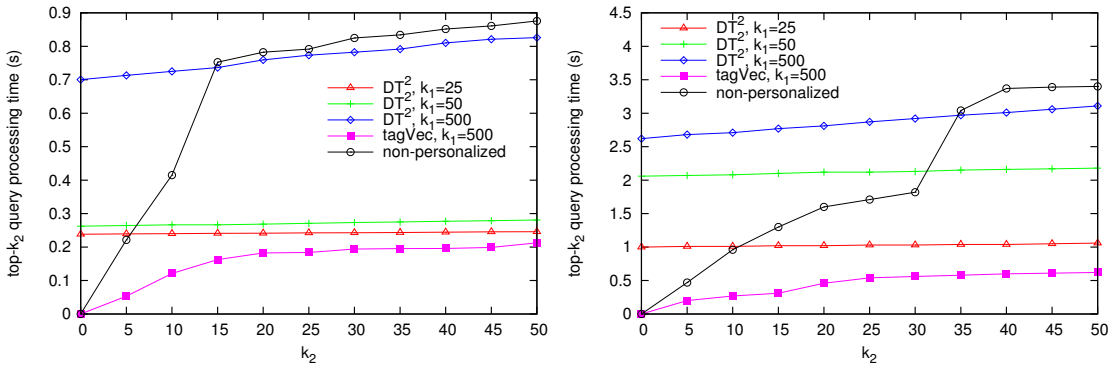
approach and the $TagVec$ off-line personalized one, when a query is issued, the system only needs to process it with the threshold algorithm by scanning the pre-computed inverted lists. Figure 3.9 summarizes the run-time performance of DT^2 with different size of personal network (k_1) compared to *TagVec* with a personal network of 500 users for each querier and the non-personalized approach when different number of items (k_2) are returned as the results.

In Figure 3.9, $k_2 = 0$ for DT^2 corresponds to the time required for selecting the top k_1 users. We observe that although the more items are returned, the longer it takes to answer the query, the query processing time in DT^2 is dominated by the user selection procedure. This confirms our choice of the simple top-k protocol for the item selection.

Comparing DT^2 to the non-personalized approach, we observe when the number of returned items is small ($k_2 \leq 5$ for *CiteULike* and $k_2 \leq 10$ for *delicious*), DT^2 requires longer time than the non-personalized approach. However, as k_2 increases, the advantage of DT^2 is gradually exhibited. Selecting 25 users to obtain the top-10 items requires 64% and 101% the time of the non-personalized approach for *CiteULike* and *delicious* respectively. If the top-20 items are required, the time counts for only 30% and 63% of the non-personalized approach. Selecting more users requires longer time but guarantees a better result quality. Selecting up to 500 users in *CiteULike*, DT^2 is still about 5% more efficient than the non-personalized approach if more than 10 items are requested. This conveys DT^2's efficiency.

The processing time of DT^2 is always longer than that of *TagVec*. This is due to the fact that the users in the personal network of *TagVec* are selected according to the querier's profile and thus contribute few relevant items for the queries that are not correlated to the profile, counting for a significant portion of the queries. The efficiency of *TagVec* is in fact obtained by sacrificing the result quality and requiring prohibitive storage, which is thus undesirable.

Note that the processing time on *CiteULike* is much smaller than that on *delicious* regardless of the used approach. As stated above, the users in *CiteULike* have more focused interests than those in *delicious*, which makes the difference between users more significant to prune the unqualified ones earlier.



(a) *CiteULike* (b) *delicious*

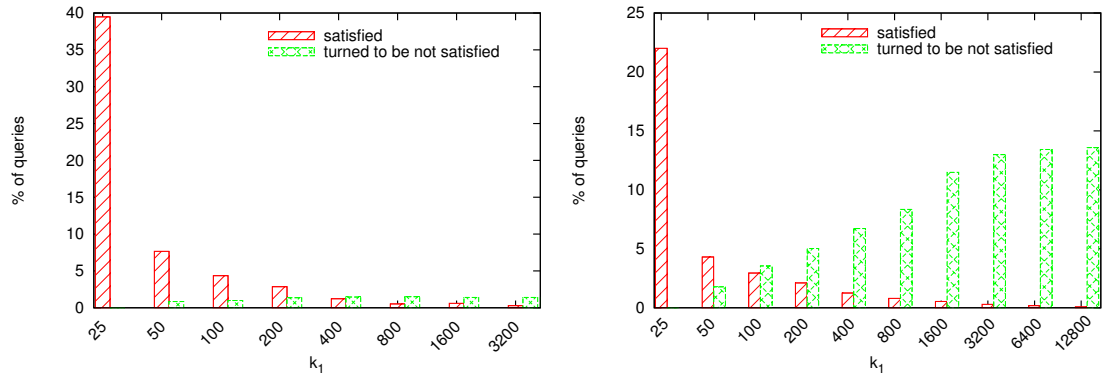
Figure 3.9: Query processing time

3.2.6.4 Fine tuning

We discuss here the impact of the size of personal network on the top k_2 results. We begin the query processing by selecting 25 users, which allows to provide relatively good results as we can see from previous evaluation. Suppose that a querier is satisfied with the results if the desired item, which was previously removed from her profile, appears in the top-10 items. This can be detected in real life by observing the click on the results for instance. If the querier is not satisfied with the current results, the system doubles the value of k_1 ($k_1^{(r+1)} = 2k_1^{(r)}$) to add users in the querier's personal network and returns the new top-10 items. This process continues until the querier is satisfied or the results cannot be further improved.

Figure 3.10 depicts the distribution of queries that can be satisfied within personal networks of different sizes (k_1). We observe that about 40% of queries in *CiteULike* and 22% of queries in *delicious* can be satisfied within a personal network of 25 users. Only less than 1% users need the opinions of more than 400 users to obtain satisfactory results. Moreover, we observe from the green bars that, instead of processing the query in a personal network with *at most* $k_1^{(r)}$ users, processing it in a personal network with $k_1^{(r+1)}$ users (k_1=$k_1^{(r+1)}$ in Figure 3.10) may lead bad results for up to 13.5% queries. These results confirm the very fact that adjusting the size of the personal network in an adaptive and incremental manner guarantees a good balance between the result quality and the processing time: most queries can be satisfied with a handful of users and thus very efficiently.

We do not mention here the time for the queries processed within the personal networks of different sizes. But as we analyzed in Section 3.2.5 and observed in Section 3.2.6.3, the latency due to adjusting the size of personal network mainly depends on the time required for sorting the items in the personal network, which is relatively small compared to the time needed for finding these users. The time is thus comparable to that of processing directly the query in a large personal network. However, the results are displayed to the querier with some time interval so that the querier will have time to assess the current results while indicating the system whether to continue.



(a) *CiteULike* (b) *delicious*

Figure 3.10: Adaptive adjustment of k_1 for top-10 items

3.2.7 Concluding remarks

The potential of personalizing the search process through social ties has been recognized and investigated quite often in the recent years. Various notions of affinities have also been developed to explore similarities between users [60, 63, 64, 66, 67, 113]. These mainly focus on the past behavior of users, such as their tagging histories [67, 63], their query histories [64, 72] or their browsing histories [60]. Although the notion of recent history was distinguished in [72] and [114], these approaches are query-oblivious and mainly achieve good results, when queries are correlated to the (recent) history.

Basically, DT^2 is the first algorithm that narrows down the top-k processing on-line. The personalization in DT^2 is achieved by capturing the similarity between users, on-line, based on both the tagging history and the query. As we have shown in the previous section, this approach performs well regardless of the correlation of the query with the querier's tagging history.

DT^2 is generic in the sense that alternative similarity measures [113], scoring functions [66, 115, 116] and underlying top-k processing sub-algorithms [47, 46] can be chosen and optimized accordingly. For instance, other metrics could be considered to compute the similarity between users, such as Jaccard similarity and Dice coefficient. Our preliminary experiments showed however the superiority of our cosine similarity metric compared to the others. Also, we considered the classical No Random Access (NRA) algorithm when selecting the most appropriate users to perform a query. Our preliminary experiments (on *CiteULike*) have indeed conveyed the fact that NRA requires much more computation to determine the similarity between users (about 18 times more partial computation).[2] An alternative approach we considered was that of [112], where the k nearest neighbors to provide tag recommendation for a resource are obtained by considering only the users who have annotated this resource. According to our experiments however, our TA-based approach revealed faster. Still, we do believe there is room for improving the (space and time) performance and result quality of DT^2, and indirectly of the on-line personalization approach. For instance, analysis on users' tagging behaviors would help understand the relationship between interest dispersion

[2] In fact, NRA cannot provide exact similarities for the top-k users, which should be thus re-computed while ranking the items in DT^2. The larger k, the more costly the additional computation will be.

and user qualification, and thus help investigate more effective similarity measures to select the top-k users for each querier's query. Also, we focused in this work on top-k processing approaches that are solely based on tags. It would be interesting to see how these can be combined with approaches that use other sources of information, such as the keywords of a paper.

3.3 DT^{2}P^2: peer-to-peer on-line personalization

In the previous section, we explored the effectiveness of on-line personalization to satisfy personalized queries, especially those reflecting the queriers' emerging interests. As we have seen, this on-line personalization can be efficiently achieved in a centralized collaborative tagging system by doing top-k twice. We are now taking a shift and investigate how the same on-line personalization can be applied in a peer-to-peer environment.

A crucial issue for the efficiency of on-line personalization is to efficiently identify the right users for answering a query once it is issued. This is achieved in DT^2 by performing a TA-style top-k processing on all the users in the system. Yet, in a peer-to-peer system, no user has such global information and each user is only aware of a small portion of other users in the system. Therefore, to perform the on-line personalization in a peer-to-peer system, the main problems we face are *(i)* which should be the set of users each user is aware of; *ii)* how can a querier efficiently identify the users for processing her query based on such partial information.

In this section, we present our algorithm DT^{2}P^2 that achieves on-line personalization for the top-k query processing in peer-to-peer systems. In DT^{2}P^2, users flexibly and efficiently self-organize into a multi-objective overlay that enables efficient top-k processing. The stable part of the overlay aims to maintain the sustainable knowledge of each user about the system, i.e., the set of users having similar tagging profiles. The flexible part of the overlay is in contrast maintained at query time to associate each querier with the users that can serve the query processing according to her hybrid interest as defined by the on-line personalization (Formula 3.1). Both parts are dynamically maintained through periodically gossiping and computing the similarity among users.

3.3.1 System design

We consider a collaborative tagging system on top of a peer-to-peer network, i.e., each user in the collaborative tagging system is associated with a peer and is characterized by her tagging profile. Users issue queries in the form of a set of tags to search for the k most relevant items.

As shown in Section 3.2.6.2, the hybrid interest that takes into account both the profile and the query outperforms the non-personalized and the off-line personalized top-k approaches. Here we apply the same on-line personalization scheme in the context of peer-to-peer systems. More specifically, the profile of a user u_o is modeled as a profile vector $\vec{p}(u_o)$ and her query $Q(u_o)$ is modeled as a query vector $\vec{q}(u_o)$ as described in Section 3.2.2. Once a user u_o issues a query $Q(u_o)$, her hybrid interest at query time is expressed with the hybrid interest vector $\vec{l}\big(u_o, Q(u_o)\big)$ with Formula 3.1.

Therefore, the problem of providing on-line personalization for a user u_o's query $Q(u_o)$ is equivalent to the problem how the user u_o can efficiently find the users sharing the most similar interests with her hybrid interest $\vec{l}\big(u_o, Q(u_o)\big)$.

3.3.1.1 System model

As users live in a fully decentralized system, each user plays several roles simultaneously: *(i)* storing and forwarding the replicas of profiles; *(ii)* collecting information and processing the query; *(iii)* keeping the connectivity of the system. These roles are achieved by each user through maintaining three data structures: an offer network, one or more demand networks and a random view. Figure 3.11 depicts the offer network and the demand network(s) associated with each user, constituting the multi-objective overlay of DT^{2}P^2. The random view is omitted as it plays the same role and ensures the connectivity of the system as that in our P4Q protocol (Section 2.4.1).

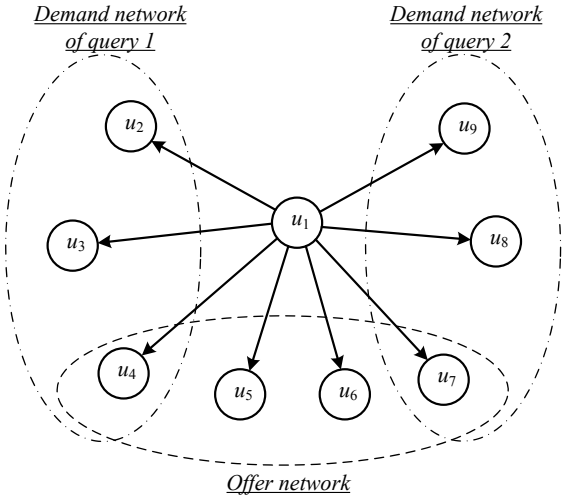


Figure 3.11: System model

Offer network The offer network of a user u_o is a set of s_o neighbors having the most similar profile with her, noted as $oNet(u_o)$. Given a user u_d, we use $Similarity_O(u_o, u_d)$ to quantify the similarity between her profile and that of u_o and determine whether she should be a neighbor of u_o in $oNet(u_o)$. This similarity is computed as the cosine of their profile vectors, i.e.,

$$Similarity_O(u_o, u_d) = cos\big(\vec{p}(u_o), \vec{p}(u_d)\big).$$

The offer network is *query-oblivious* and each user has only one offer network. Considering that users often have some continuity in their tagging behaviors, the offer networks built based on the tagging profiles are relatively stable. Therefore the offer networks form the backbone of the entire system.

For each neighbor in the offer network, both her profile vector and her whole profile are stored along with her contact information. The offer network serves to store the replicas of the neighbors' profiles. These profiles are also used to process the top-k queries.

Demand network The demand network of a querier u_o for her query $Q(u_o)$ consists of s_d neighbors that u_o relies on to answer $Q(u_o)$. Let $\vec{l}\big(u_o, Q(u_o)\big)$ be the hybrid interest vector of u_o for

query $Q(u_o)$, the similarity of a user u_d to the querier u_o and her query $Q(u_o)$ is measured by

$$Similarity_D\big(\langle u_o, Q(u_o)\rangle, u_d\big) = cos\big(\vec{l}\big(u_o, Q(u_o)\big), \vec{p}(u_d)\big).$$

The s_d users having the highest $Similarity_D\big(\langle u_o, Q(u_o)\rangle, u_d\big)$ form the demand network of u_o for the query $Q(u_o)$. This demand network is equivalent to the personal network of each querier in DT^2 (Section 3.2.3.3). Yet, without global information, it is difficult for u_o to identify her ideal[3] demand network directly. Instead, u_o gossips with other users in the system to gradually improve her demand network. We use $dNet_c\big(u_o, Q(u_o)\big)$ to denote the demand network of u_o at the cth gossip cycle for answering her query $Q(u_o)$.

The demand network is *query-specific* and is maintained for each query. Once a query is issued, the profile vector and the contact information of each neighbor is *temporarily* maintained at each cycle until the query is resolved. As we will see, the offer networks help the queriers to efficiently discover qualified neighbors for their demand networks by offering good candidates.

Random view Each user also maintains a set of s_r random neighbors, whose profile vectors and contact information are stored. These users forms her random view, denoted as RPS.

The maintenance of the offer network and the random view follows the same scheme as the lazy mode in P4Q and leverages the same convergence property. The only difference is that instead of gossiping the profile digests in Bloom filters, the profile vectors are first transmitted to determine whether a user is a neighbor. We omit this part and more details can be referred to in Section 2.4.2.1.

3.3.1.2 Querying model

In DT^{2}P^2, a query is processed within the demand network of the querier for it. When a query is issued, the querier gradually build her demand network by gossiping with her neighbors. The results are refined by processing the query within each intermediate demand network until the querier is satisfied or the demand network converges.

Finding the appropriate neighbors for the demand network at query time can be considered as the querier brings her query and *virtually "moves"* in the system from her stable position defined by her offer network to the query effective position defined by her demand network.

Figure 3.12 illustrates the movement of a querier with her query from her offer network toward her demand network. The querier u_1 and her query can be considered as a *virtual user* v_1, who joins the system at query time in the same position as u_1 and whose preferences are characterized by the hybrid interest of u_1. v_1 (u_1) first gossips with her neighbor u_3 to discover a new neighbor u_6 for her demand network, which corresponds to one step of movement. Then she gossips with her neighbor u_2 and this allows her to arrive a new position surrounded by u_4 and u_6. The movement stops when the demand network converges, i.e., u_4 and u_6 appear to be the most qualified neighbors for the demand network.

Here we use the notion of *virtual user* as the querier always stays in the stable position defined by her offer network. The underlying intuition is that if two consecutive queries are not correlated to each other, once the querier moves to a new position, where she is surrounded by the neighbors in the demand network of the first query, it might be difficult for her to find good neighbors for the

[3]By ideal, we mean the demand network obtained using the entire set of profiles in the system.

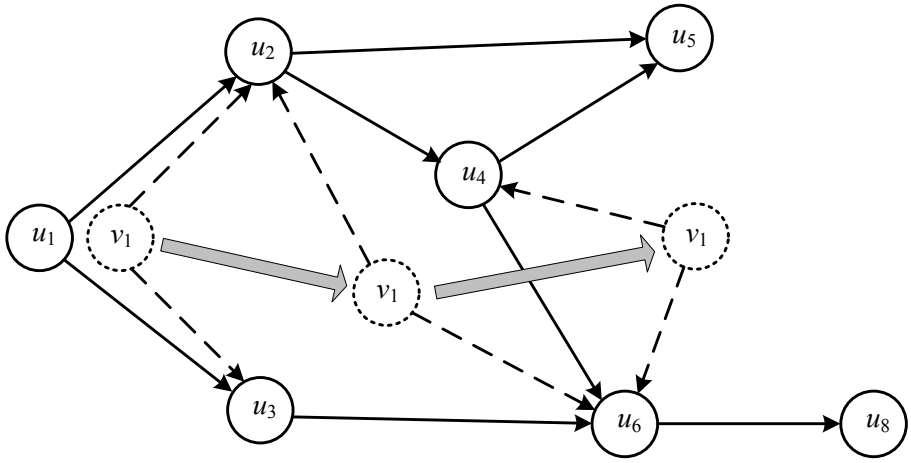


Figure 3.12: Querying model

demand network of the second query. In contrast, the offer network is built according to her tagging profile, which is generally correlated to her queries.

We details in the following how the demand network is refined at query time and how the query is processed accordingly.

3.3.2 Refining the demand network

When a query $Q(u_o)$ is generated, the querier u_o first considers her offer network as her initial demand network, and locally processes the query with the profiles of her neighbors in this network. As we said, this is equivalent that a virtual user joins the system in the same position defined by u_o's offer network. Considering that a user often searches for the topics she is interested in, it is highly probable there is someone in her offer network who is also interested it the same topic. If the querier u_o is not satisfied with the current results, she begins to gossip with her neighbors to refine this demand network. At each cycle, u_o gossips with a neighbor from her demand network of this query. The neighbor u_d ($u_d \in dNet_0\big(u_o, Q(u_o)\big)$) whose profile vector is the most relevant to the query and who does not gossip with u_o recently is selected as gossip destination. The relevance of the profile vector to the query is measured by the cosine similarity between the two vectors, i.e., $cos\big(\vec{p}(u_d), \vec{q}(u_o)\big)$.

A timestamp is assigned to each neighbor in the demand network to adjust their order in gossip. The initial value of each neighbor's timestamp is 0. When a neighbor u_d with timestamp $T(u_d)(T(u_d) \geq 0)$ is selected as gossip destination, her timestamp is set to $T(u_d) - s_d$, where s_d is the size of the demand network. If some new users are discovered during the gossip and added to the demand network, their timestamps are set to 0. The timestamp of each neighbor in the refined demand network is then increased by 1. The querier only selects the gossip destination among the neighbors having no negative timestamps, which ensures that the same user with high relevance to the query would not be successively selected. In contrast, waiting at least s_d cycles before being selected again allows the corresponding neighbor to capture the changes in the system if any.

If no neighbor has a positive similarity to the query, a random neighbor with oldest timestamp will be selected as gossip destination. Similarly, if more than one neighbor has the same similarity,

a random one with oldest timestamp will be selected among them.

When receiving the gossip message containing the query from the querier, the user u_d sends a subset of the profile vectors in her offer network $oNet(u_d)$ that contains at least one common tag with the query to querier. The underlying intuition is that the querier u_o gossips with u_d supposing she is knowledgeable in the tags related to the query as indicated by her profile vector. In the meantime, the neighbors in u_d's offer network are those having similar tagging profiles with her and should also be knowledgeable in the query tags. Fetching such information from the gossip destination allows the querier to quickly discover new qualified neighbors for her demand network. Moreover, if a user's profile does not contain any tag in the query, she will not contribute to the query results. It is thus useless to recommend her profile to the querier for feeding her demand network. The s_d users having the highest similarity score $Similarity_D\big(\langle u_o, Q(u_o)\rangle, u_d\big)$ among the users in u_o's current demand network and those received from u_d are kept to form a refined demand network. Then the query is processed with the information in the refined demand network to obtain the query results at this cycle.

This process continues for each gossip cycle until the user feels satisfied with the top-k results or the demand network converges to a stable state, i.e., the neighbors remain the same during s_d cycles. In this case, keeping gossiping with these neighbors will not bring new neighbors if no changes occur in the system. Note that it does not necessarily mean that the demand network is the same as that obtained using the hybrid interest model over all the users in the system. In fact it is still possible to further improve the result quality even if the demand network converges. It is sufficient to use some other mechanisms to choose gossip destination when the demand network converges, e.g., gossiping with a user from the random view who has the query tags in her profile vector. Algorithm 3.4 depicts the gossip procedure for refining the demand network.

Algorithm 3.4 Per cycle demand network refinement

1. **Input:** $Q(u_o)$, $oNet(u_o)$ and $dNet_{c-1}\big(u_o, Q(u_o)\big)$
2. **Output:** $dNet_c\big(u_o, Q(u_o)\big)$
3. **if** $dNet_{c-1}\big(u_o, Q(u_o)\big)$ is empty **then**
4. $\quad dNet_{c-1}\big(u_o, Q(u_o)\big) \leftarrow oNet(u_o)$
5. **end if**
6. select u_d from $dNet_{c-1}\big(u_o, Q(u_o)\big)$ with $T(u_d) \geq 0$ and max$\{cos\big(\vec{q}(u_o), \vec{p}(u_d)\big)\}$
7. $T(u_d) \leftarrow T(u_d) - s_d$
8. gossip with u_d and receive profile vectors $\vec{p}(u_l)$ in $oNet(u_d)$ from u_d
9. **for** each received $\vec{p}(u_l)$ **do**
10. $\quad Similarity_D\big(\langle u_o, Q(u_o)\rangle, u_l\big) \leftarrow cos\big(\vec{l}\big(u_o, Q(u_o)\big), \vec{p}(u_l)\big)$
11. $\quad$ add u_l to $dNet_{c-1}\big(u_o, Q(u_o)\big)$
12. $\quad T(u_l) \leftarrow 0$
13. **end for**
14. keep s_d users with largest $Similarity_D\big(\langle u_o, Q(u_o)\rangle, u_d\big)$ in $dNet_{c-1}\big(u_o, Q(u_o)\big)$ to form $dNet_c\big(u_o, Q(u_o)\big)$
15. **for** each u_l in $dNet_c\big(u_o, Q(u_o)\big)$ **do**
16. $\quad T(u_l) \leftarrow T(u_l) + 1$
17. **end for**

3.3.3 Processing the query

A query is processed in collaboration with the querier and other users who gossip with her. We here describe how the querier processes the query based on the refined demand network of the query at each cycle and how the other users contribute to the query processing.

As mentioned above, before gossiping the query, the querier u_o first computes the top-k results based on the profiles in her initial demand network (offer network). The top-k items are obtained by directly iterating over these profiles. All the items tagged by the same user u_l with the tags in the query form a partial result list for this query, i.e.,

$$List\big(u_l, Q(u_l)\big) = \{\langle i_1, Score\big(u_l, Q(u_o), i_1\big)\rangle, ..., \langle i_1, Score\big(u_l, Q(u_o), i_m\big)\rangle, ...\}.$$

The score of each item in the list $List\big(u_l, Q(u_l)\big)$ is the number of tags in the query $Q(u_i)$ that are used by u_l to tag it, i.e.,

$$Score\big(u_l, Q(u_o), i_m\big) = |\{\langle u_l, i_m, t_n\rangle | t_n \in Q(u_o)\}|. \tag{3.7}$$

Then the top-k items can be obtained by merging all the lists and bubbling up the k items with the highest relevance scores. The relevance score of an item i_m to the querier u_o's query $Q(u_o)$ at the cth cycle is computed with as

$$Score\big(Q(u_o), i_m\big) = \sum_{u_d \in dNet_c\left(u_o, Q(u_o)\right)} Score\big(u_d, Q(u_o), i_m\big) Similarity\big(\langle u_o, Q(u_o)\rangle, u_l\big), \tag{3.8}$$

Formula 3.8 is the same as Formula 3.4 in DT^2 to compute the relevance score. The only difference is that the query is processed in the intermediate demand network $dNet_c\big(u_o, Q(u_o)\big)$ instead of the personal network in DT^2.

Once the demand network $dNet_c\big(u_o, Q(u_o)\big)$ is refined and some neighbors in the former demand network $dNet_{c-1}\big(u_o, Q(u_o)\big)$ are replaced by the new neighbors, the querier u_o contacts the user u_d who is gossiping with her to ask for her contribution. Note that during the demand network refinement phase, only the profile vectors are sent to the querier u_d in order to not overload the system by transmitting useless information. This is because the profiles of the users that cannot be u_o's neighbors in $dNet_c\big(u_o, Q(u_o)\big)$ would not be used for the query processing. u_d computes the partial result lists as presented above with Formula 3.7 using the profiles of u_o's new neighbors and sends them back to the u_o. Merging these lists before sending to the querier may reduce the transmission cost as there are common items in these lists, however, it would be difficult for the querier to restore the list of any single user when the demand network evolves.

The partial result lists used in the previous cycle $(c-1)$ are temporarily stored by the querier to serve the query processing in the refined demand network $dNet_c\big(u_o, Q(u_o)\big)$ if there are some common neighbors in $dNet_c\big(u_o, Q(u_o)\big)$ and $dNet_{c-1}\big(u_o, Q(u_o)\big)$. This avoids consecutively transmitting the same lists. Upon receiving the new lists, the querier merges all the lists corresponding to the refined demand network $dNet_c\big(u_o, Q(u_o)\big)$ and computes the top-k items. Algorithm 3.5 depicts the collaborative top-k query processing.

3.3.4 Boosting the query processing

Once a query is generated, the querier considers her offer network as the initial demand network for this query. After the query is locally processed, the querier begins gossiping by selecting a

Algorithm 3.5 Per cycle top-k processing

1. **Input:** $Q(u_o)$, $oNet(u_o)$, $dNet_{c-1}\big(u_o, Q(u_o)\big)$ and $dNet_c\big(u_o, Q(u_o)\big)$
2. **Output:** top k items
3. **for** each u_l in $dNet_c(u_o, Q(u_o))$ **do**
4. **if** u_l not in $dNet_{c-1}\big(u_o, Q(u_o)\big)$ and not in $oNet(u_o)$ **then**
5. ask gossip destination u_d for $List\big(u_l, Q(u_o)\big)$
6. **end if**
7. **end for**
8. **for** each i_m in $List\big(u_l, Q(u_o)\big)$ $\big(u_l \in dNet_c\big(u_o, Q(u_o)\big) \wedge u_l \notin oNet(u_o) \wedge u_l \notin dNet_{c-1}\big(u_o, Q(u_o)\big)\big)$ **do**
9. compute $Score\big(Q(u_o), i_m\big)$
10. add $\big\langle i_1, Score\big(u_l, Q(u_o), i_m\big)\big\rangle$ to heap
11. **end for**
12. rank the first k items in heap
13. return the first k items in heap as top k items

neighbor from this demand network, which allows her to discover new neighbors and further refine her demand network. Here the offer network serves as the starting point of the querier's movement towards the good neighborhood and results.

However, if a query is not very correlated to the querier's profile, it would take more time for the querier to find good neighbors for its demand network through gossiping. In an extreme case where none of the neighbors in the querier's offer network has used the query tags, the querier can only gossip with a random neighbor from them. There is no guarantee that the selected user has used these query tags so that she can hardly contribute to the query processing or bring new neighbors for the current query's demand network. As a consequence, the querier has to gossip with several other neighbors before finding a good neighbor to process her query. In contrast, if a similar query was previously issued by the querier, better results should be expected if the query is first processed within the demand network of this query. Moreover, gossiping with users from this demand network would also facilitate the discovery of new neighbors for the current query's demand network. This is because the neighbors in the offer network of the gossiped user are also likely to have profiles correlated to the current query. In fact, when the query is correlated to the querier's profile, if there is a demand network previously built for a similar query, starting from this demand network would also help the querier to find a better neighbor to gossip with as the users in this network are selected with more emphasis on the similar query.

Motivated by these observations, we argue that if each user maintains a cache containing well selected demand networks of her previous queries, the query results would be more efficiently improved while she is "moving" towards the desired demand network. The cached demand networks can serve as a shortcut on the path to the good neighborhood for answering a query. This can be considered as the virtual user corresponding to the querier does not join the system always in the same position but in a better position memorized by the querier. For instance, in Figure 3.13, the user u_1 has two demand networks in her cache, which define the position p_1 and p_2 in the system. If the virtual user v_1 starts her movement from p_1, only one gossip would be enough to arrive the same position where she needed two gossips to arrive without the cache.

We use $Cache(u_o)$ to denote the cache of a user u_o. Each entry in the cache corresponds to a query along with its demand network and the timestamp when it is added to the cache in form

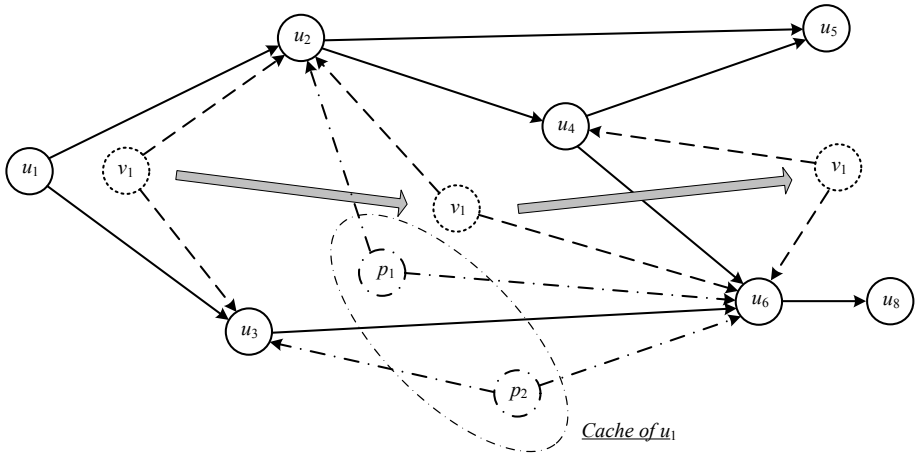


Figure 3.13: Querying with cache

of $\langle Q_j(u_o), dNet(u_o, Q_j(u_o)), time(Q_j(u_o))\rangle$. $dNet(u_o, Q_j(u_o))$ is the final demand network where $Q_j(u_o)$ was processed for the last time. Each cache contains at most s_c entries.

In the following we explain *(i)* how the demand networks of previous queries are selected to feed the cache, *(ii)* how the cache is used to serve the query processing and *(iii)* how to cope with the impact of user departure on cache.

Managing the cache Caching is a useful technique for search engines that enables a shorter average query response time. With the cache of demand networks, it reduces the processing time by reducing the number of gossips before satisfying a query and thus reduces the overall amount of used bandwidth. In general, the probability that a query can benefit from the cache is higher if more entries are kept in the cache. However, different from centralized search engine, which can approximately have infinite caches as they store a large number of entries using RAM and disk, the storage that an individual user in a peer-to-peer system can allocate to the cache highly depends on her own capability and thus should be limited. In fact, even if a user can actually maintain a large number of entries in the cache, it would not be a preferable choice in a highly dynamic environment where the users leave and join the system and continue to change their interests by generating and tagging new contents.

To maximize the utility of the cache and minimize the negative impact of dynamics, we keep in the cache the demand networks of the queries whose union covers the maximum number of tags. We also use a TTL (time-to-live) value to control the age of each entry and only the demand networks of queries issued within the TTL are cached.

We prefer the demand networks of queries that maximize the coverage of tags rather than those are frequently issued to feed the cache for the following reasons: *(i)* the user frequently asks for something mainly because she is actually interested in it. In this case, the offer network composed of users having similar profiles should already contain some good candidates to gossip with and provide satisfactory results; *(ii)* different from in centralized search engines where the same query may be generated by different users, in distributed systems, a user rarely sends the same query. Thus the cache does not aim to directly provide the perfect results for a query but to offer some better candidates (than those in the offer network) to gossip with. Therefore, there is no need to keep all

the frequently asked queries in the cache as the demand networks of similar queries would consist of common users.

We use a cache vector, noted as $\vec{c}(u_o)$, to denote the tags covered by the queries in the cache $Cache(u_o)$. We also use the notation $C_Q(u_o)$ to represent the set of queries whose demand networks are cached. Each element in the cache vector corresponds to a tag that appears in at least one query in the cache and its value is the total times it appears in the cached queries, i.e.,

$$\vec{c}(u_o) = \big[w_c(t_1, u_o), w_c(t_2, u_o), ..., w_c(t_n, u_o)\big],$$

where $w_c(t_n, u_o) = \mid \{Q_j(u_o) \mid Q_j(u_o) \in C_Q(u_o) \wedge t_n \in Q_j(u_o)\} \mid$. We keep the queries that maximize the dimension of the cache vector in the cache.

An entry is considered expired and discarded from the cache if the difference between current time and the time it was added in the cache is larger than the TTL. It is crucial to carefully select the TTL value to guarantee the quantity of available information in the cache and the freshness of such information. Too small TTL would cause the less active users to have few entries in their caches while too large TTL would hurt the freshness of the entries. In the latter case, the query may be guided to the users that are no longer interested in the searched topics or even be lost due to the departure of the concerned users.

The queries of an individual user arrive in the system consecutively as a user rarely sends several queries at the same time. So each user updates her cache by sequentially evaluating each query once it is answered and replacing an old entry with the new one if necessary. Algorithm 3.6 depicts how a user manages her cache. We use $dim\big(\vec{c}(u_o)\big)$ to denote the dimension of the cache vector $\vec{c}(u_o)$.

Selecting the initial demand network When a query is issued, if the querier has some previously computed demand networks in her cache, instead of directly considering her offer network as the initial demand network, she selects from her offer network and cached demand networks the one that is the most beneficial for the current query.

As presented above, the demand network is query-specific. The demand network of a query is built based on the querier's hybrid interest vector, which is a combination of her profile and her query. For two queries of the same user, the interest vectors differ from each other only in the query and the two corresponding weights (Formula 3.1), which are in fact determined by the query. The more similar two queries, the more similar the corresponding interest vectors. Thus the more similar their demand networks are. Therefore, the demand network of the query that is the most similar to the current query is preferential to be selected as the initial demand network. The similarity between two queries are measured by the cosine similarity between their query vectors, i.e.,

$$Similarity\big(Q_i(u_o), Q_j(u_o)\big) = cos(\vec{q}_i(u_o), \vec{q}_j(u_o)).$$

Yet, for the query that is very correlated to the profile, the offer network should be good enough as the initial demand network. In contrast, even the query corresponding to the best cached demand network may have few common tags with the current query. To avoid misleading the query processing, the final decision is made by comparing the similarity between the current query and the profile to that between the current query and the query of the most qualified demand network in the cache. If the former is larger, the query processing starts from the offer network as introduced above. Otherwise, it starts from the selected demand network in the cache.

Algorithm 3.6 Per user cache management

1. **Input:** $Q_{current}(u_o)$, $Cache(u_o)$ and $\vec{c}(u_o)$
2. **Output:** updated $Cache(u_o)$
3. **for** each entry $\langle Q_j(u_o), dNet\big(u_o, Q_j(u_o)\big), time\big(Q_j(u_o)\big)\rangle$ in $Cache(u_o)$ **do**
4. **if** $time\big(Q_{current}(u_o)\big) - time\big(Q_j(u_o)\big) \geq TTL$ **then**
5. remove the entry from $Cache(u_o)$
6. remove $Q_j(u_o)$ from $\vec{c}(u_o)$
7. **end if**
8. **end for**
9. **if** $\mid Cache(u_o) \mid < c$ **then**
10. add $\langle Q_{current}(u_o), dNet\big(u_o, Q(u_o)\big), time\big(Q_{current}(u_o)\big)\rangle$ to $Cache(u_o)$
11. add $Q_{current}(u_o)$ to $\vec{c}(u_o)$
12. **else**
13. **for** each $\langle Q_j(u_o), dNet(u_o, Q_j(u_o)), time\big(Q_j(u_o)\big)\rangle$ in $Cache(u_o)$ **do**
14. $\vec{c}_j(u_o) \leftarrow$ remove $Q_j(u_o)$ from $\vec{c}(u_o)$
15. $\vec{c}_j(u_o) \leftarrow$ add $Q_{current}(u_o)$ to $\vec{c}(u_o)$
16. **if** $dim\big(\vec{c}(u_o)\big) \leq dim\big(\vec{c}_j(u_o)\big)$ **then**
17. add $\langle Q_j(u_o), \vec{c}_j(u_o)\rangle$ to $candidates$
18. **end if**
19. **end for**
20. **if** $candidates$ not empty **then**
21. $Q_{max}(u_o) \leftarrow Q_j(u_i)$ with $max\big\{dim\big(c_j(u_o)\big)\big\}$ then $min\big\{time\big(Q_j(u_i)\big)\big\}$
22. **end if**
23. replace the entry of $Q_{max}(u_o)$ by that of $Q_{current}(u_o)$ in $Cache(u_o)$
24. $\vec{c}(u_o) \leftarrow \vec{c}_{max}(u_o)$
25. **end if**
26. return $Cache(u_o)$

Processing the query from a cached demand network Once a querier selects her initial demand network for a query, the query is first processed within this network with the profiles of all the neighbors in it. Different from the offer network, where the whole profiles of the neighbors are locally stored by each, the demand networks in the cache only contain the profile vector of each neighbor. For the neighbors that are also in a querier's offer network, the partial result lists are locally computed for each neighbor. Then the querier forwards the query to the other neighbors in her demand network to fetch their partial result lists for this query. In this way, the user obtains the first top-k items by merging all the partial result lists. If the user is not satisfied with the results, she begins to gossip with the neighbors in her initial demand network and gradually improve the result quality as presented in Section 3.3.2 and Section 3.3.3.

Handling churn Once the demand network of a query is added to the cache, the neighbors in it remain the same until it is expired and removed from the cache. However, peer-to-peer systems are dynamic in the sense that users frequently leave and join the system. As presented above, the querier needs to contact the neighbors in the selected demand network to process her query. If some users leave, the information in their profiles would be temporarily unavailable. This is unavoidable to decrease the benefits of starting the query processing from a cached demand network. Moreover, the leaving users can no longer gossip with the querier. In contrast, the offer network is periodically maintained and once some neighbors leave, they can be quickly detected and replaced by active

users as in many gossip-based protocols.

To minimize the influence of churn, once the leaving users are detected while fetching the partial result lists from them, they are removed from the querier's initial demand network of the current query. The gossip destination is then selected from the active neighbors. As the profile vectors received from the gossip destination are those in her offer network, they are more likely to be active and can help the querier quickly repair her demand network. In the extreme case where all the neighbors in the select demand network have left, the querier turns back to her offer network for the query processing.

Note that even if some neighbors in the selected demand network have left, the network is not updated in the cache. The reason is that the remaining users would still help the query to quickly reach the user that is knowledgeable in the searched tags if the current query is similar enough to the query of the selected demand network. Otherwise, starting from the offer network will not be influenced by the incomplete demand networks. This also avoids the maintaining cost and the leaving users can be efficiently replaced at query time as we will see in Section 3.3.5.2. In fact, when the leaving users come back to the system, they can continue to serve the query processing.

3.3.5 Experimental evaluation

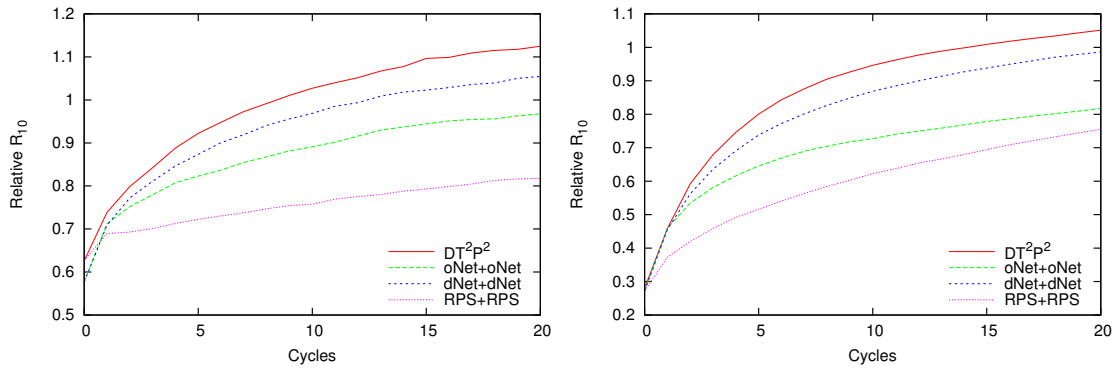
We evaluate in this section the ability of DT^{2}P^2 to enable on-line personalization in peer-to-peer environments. Section 3.3.5.1 assesses the efficiency of DT^{2}P^2. Section 3.3.5.2 emphasizes the impact of cache on the efficiency, especially when the users' interests change and users leave. We finally introduce the cost of DT^{2}P^2 in terms of bandwidth and storage in Section 3.3.5.3.

3.3.5.1 Efficiency of query processing

We evaluate the efficiency of DT^{2}P^2 using real traces from *CiteULike* and *delicious*. We use exactly the same subsets of data to establish the experimental systems as described in Section 3.2.6.1. In other words, the same users as those in DT^2 are used for the experiments. Each user has the same profile and issues the same query. The objective here is to measure how efficiently DT^{2}P^2 allows each querier to obtain the ideal top-k results defined by our on-line personalization. As a result, we use the *relative recall* at each cycle to assess how the result quality approximates the ideal one. Relative recall is computed by dividing the recall at each cycle by the (ideal) recall obtained in DT^2. The larger the value, the better the result quality. Relative recall is 1 means the same level of result quality as in DT^2 is obtained.

To convey the rationality of DT^{2}P^2 as well as its efficiency, we compare it against 3 other candidate systems where each user is also equipped with her offer network, demand networks and random view. Yet, each user solely relies on one of these networks to select the gossip destination and compose the gossip message to refine the demand network for a given query.

Figure 3.14 compares the top-10 results obtained with the 4 different strategies in both traces. The trends exhibited in different top-k results are quite similar so that we focus in this section on the top-10 results. "dNet+dNet" corresponds to the case where the querier gossips with a neighbor from her demand network as in DT^{2}P^2 but fetches the neighbors from the demand network of the gossiped user instead. In contrast, with "oNet+oNet", the querier always gossips with a neighbor from her offer network and fetches the the neighbors from the offer network of the gossiped user to refine her demand network. This strategy should be very efficient for the queries that are closely correlated to



(a) *CiteULike* (b) *delicious*

Figure 3.14: Efficiency of query processing

the querier's profile. "RPS+RPS" can be considered as the worst case where the querier gossips in a completely random manner.

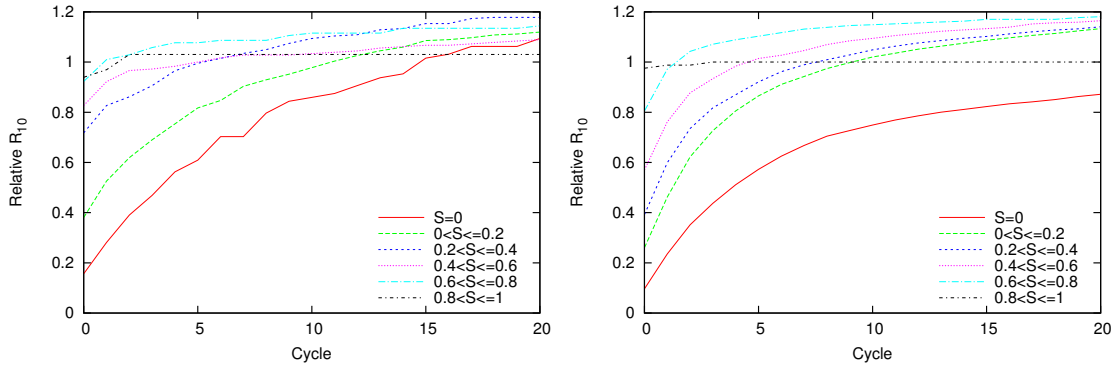
We observe from this figure that 9 cycles and 15 cycles are enough to obtain the same top-10 results as in DT^2 on *CiteULike* and *delicious* respectively. Note that the relative recall continues to improve and surpasses 1 after certain cycles. This is due to the fact that the demand network of a query is refined gradually and the query is processed with each intermediate network. As a result, more users' profiles (than the pre-defined size of demand network) are actually used to process the query. This leads to better results than those obtained in the ideal demand network with fixed number of similar neighbors.

The difference between DT^{2}P^2 and "oNet+oNet" confirms that the neighbors in the demand network are more likely to provide, through gossiping, other good users for refining the demand network. While fetching the profiles from the offer network of the gossiped users allows the querier to discover more good neighbors at once for her demand network as shown by the difference between DT^{2}P^2 and "dNet+dNet".

Figure 3.15 shows the result improvement with DT^{2}P^2 for different queries. Queries are grouped according to their similarities with the querier's profile, noted as S. We observe that the more the query is similar to the profile, the faster the relative recall reaches 1 during the gossip. Encouragingly, regardless of the similarity, the ideal result quality of our on-line personalization can be obtained within 15 cycles. The improvement is more significant for the queries that are less correlated to the querier's profile. This further confirms that our on-line personalization is very effective to cope with emerging interests of the queriers.

3.3.5.2 Effectiveness of cache

Dataset and setup We evaluate the benefit of using cache in DT^{2}P^2 with the trace from *delicious*. The *CiteULike* trace we used does not contain any information about the occurrence time of each tagging action as well as the arriving time of each query. It is thus unsuitable to mimic our dynamic management of the cache. In contrast, in the *delicious* trace, each tagging action is associated with the date indicating when it occurred.



(a) *CiteULike* (b) *delicious*

Figure 3.15: Efficiency of query processing

We focus on the experiments the tagging actions occurred from 2007-01-01 to 2008-12-31. We use the tagging actions from 2007-01-01 to 2008-06-30 to build the profile of each user and the tagging actions from 2008-07-01 to 2008-12-31 to generate the queries for each user. We randomly pick $10,000$ users, who has at least 10 tagging actions in her profile and at least 2 queries, from the entire *delicious* trace we have (Section 3.2.6.1). The queries for each user are generated by picking each item tagged by this user from 2008-07-01 to 2008-12-31 and forming the corresponding query with the tags used by this user to tag that item. The last query of each user is used to evaluate the quality of the top-10 results and all the previous ones are used to gradually feed the cache.

As described above, only the queries issued at most TTL days prior to the current query (and their demand networks) are kept in the cache. Figure 3.16 depicts the distribution of queries that can be used to feed the cache given different TTL in our dataset. As expected, the large the TTL, the more queries can be used to feed the cache of each user. As the dataset we use is relatively sparse, for $TTL = 30$ days, only 18% users have more than 10 queries issued prior to the current query. within 30 days. We thus focus in our experiments on $TTL = 180$ days, which guarantees that 65% users have more than 10 candidate queries to feed their caches.

Result quality We evaluate the effectiveness of the strategy used in DT^{2}P^2 to feed the cache by comparing the gain in recall at each cycle to 2 other strategies. The first is the well-known LRU, which in our case only keeps the most recently seen queries and their demand networks in the cache. The second is referred to as $LeastRelevant$. It keeps the queries whose cosine similarity with the querier's profile are minimum in the cache. The underlying intuition is that the cache can serve as a complement if the current query is not correlated to the querier's profile. Otherwise, the offer network of the querier should be good enough to act as the initial demand network. We define the gain in recall at the cth cycle as

$$Gain(c) = \frac{R_k(c)}{R_k^0(c)},$$

where $R_k^0(c)$ is the recall at the cth cycle if no cache is used, where the querier always considers her offer network as her initial demand network to process a query. The larger $Gain(c)$, the better the

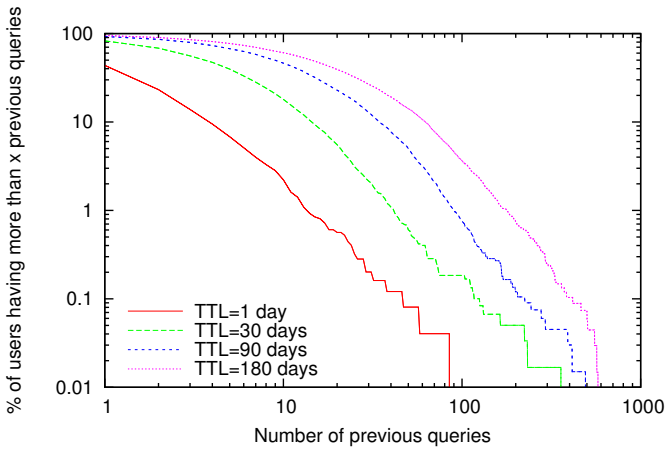


Figure 3.16: Distribution of candidate queries with different TTL

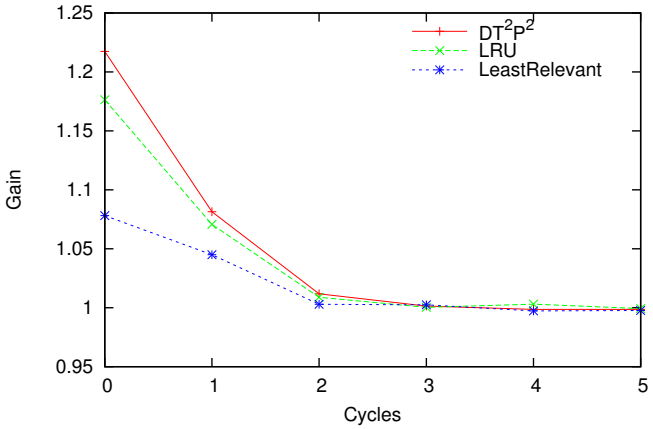


Figure 3.17: Effectiveness of DT^{2}P^2 cache

result quality. $Gain(c) = 1$ means the quality of the top-k results at the cycle c is the same as no cache is used.

Figure 3.17 compares these 3 strategies. The size of the cache is set to 5 so that different queries can be selected to feed the cache according to different strategies. The 0th cycle corresponds to the query processing within the initial demand network. Using cache significantly improves the recall at the first 2 cycles. The gain decreases as the cache only serves as the initial demand network for certain queries. During the gossip, more qualified neighbors can be gradually identified and added to the refined demand network regardless of its initial state. As a result, after the 3rd cycle, the result quality improves in almost the same rate as no cache is used. Yet, the cache allows enhancing the user experience by accelerating the query processing at the beginning.

Figure 3.18 further evaluates the impact of cache size on the query processing. Not surprisingly,

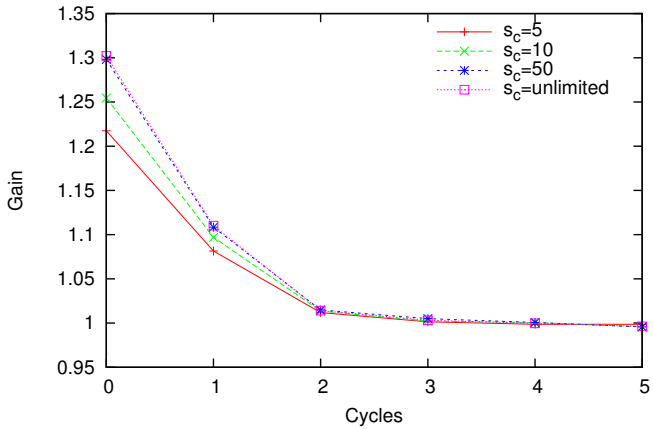


Figure 3.18: Impact of cache size

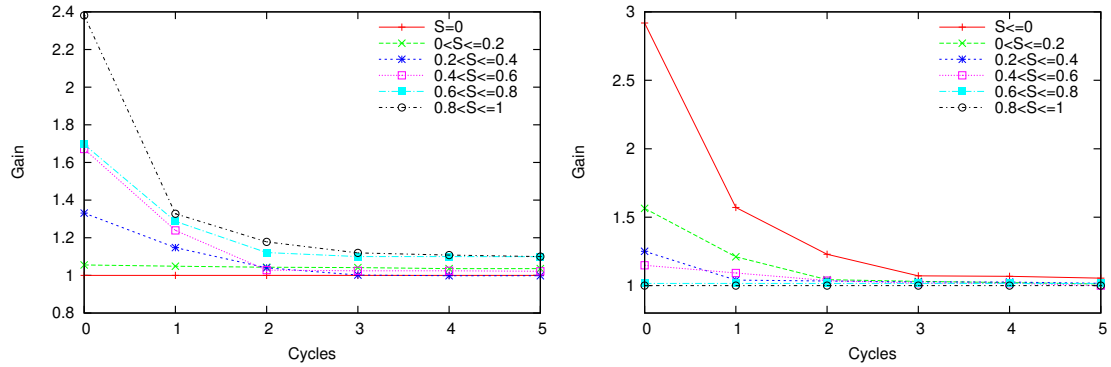
the larger the cache, the better the result quality in the initial demand network and the faster the result quality improves during the first 2 cycles. However, as shown in Figure 3.16, the number of candidate queries to feed the cache decreases quickly with the size of the cache. This explains why the difference between difference sizes of the cache is relatively small. In the following experiments, we focus on the setting with cache size 5.

To understand the rationality underlying the initial demand network selection mechanism in DT^{2}P^2, we are now interested in the impact of the query similarity on the efficiency of the query processing. We group the queries that benefit from the cached demand network according to their similarity with the corresponding selected queries or with the querier's profile respectively. Figure 3.19 (a) shows that the more the current query is similar to the selected query, the better the result quality in the initial demand network and the faster the result quality improves. This conveys that it is reasonable to choose the demand network of the most similar query as the initial demand network. In contrast, Figure 3.19 (b) reveals that the less a query is similar to the querier's profile, the more it can benefit from the cache. This result is confirms our expectation by using the cache.

Impact of dynamics on cache As the users in a peer-to-peer system is very dynamic in the sense of leaving the system and changing their tagging profiles, we are now interested in how such dynamics influence the query processing, especially when the cache is used.

User departure Users in a peer-to-peer system can leave and join the system at any time. The left of users would make their profiles unavailable for the cached demand networks where they are involved in. As the offer network of each user is periodically maintained, the user departure mainly influences the query processing when a cached demand network is selected as the initial demand network as described in Section 3.3.4.

We assume that a random portion of users (p) leave the system simultaneously. We observe from Figure 3.20 that the more users leave the system, the more the result quality degrades in the initial demand network (Cycle 0). Yet, if only 10% users leave, the result quality is still better than that if no cache is used ($Gain(0) > 1$). This is because the proportion of leaving users in each cached



(a) Similarity w.r.t. selected query

(b) Similarity w.r.t. profile

Figure 3.19: Impact of query similarity

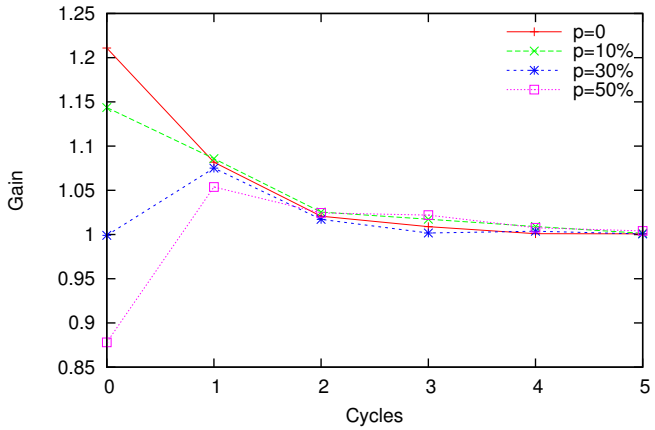


Figure 3.20: Impact of user departure

demand network is comparable to the overall proportion of leaving users. As a result, the large proportion of remaining users in each demand network can provide enough information to process the query. Even if the left of 50% users significantly degrades the result quality, after the 1st cycle of gossip, the leaving users are replaced by the active ones. Since then the result quality continues improving as no user leaves.

Interest change The tagging profile of a user in collaborative tagging system evolves when she tags new items. The offer network evolves accordingly by replacing some neighbors with more qualified ones that reflect the new trends exhibited in her new profile. Yet, the demand networks in the cache do not change until it is expired or replaced. When a user issues a query, we wonder whether it is better to consider her offer network or a probably "out-of-date" demand network as her initial demand network for this query. To this end, we evaluate the impact of the interest change on

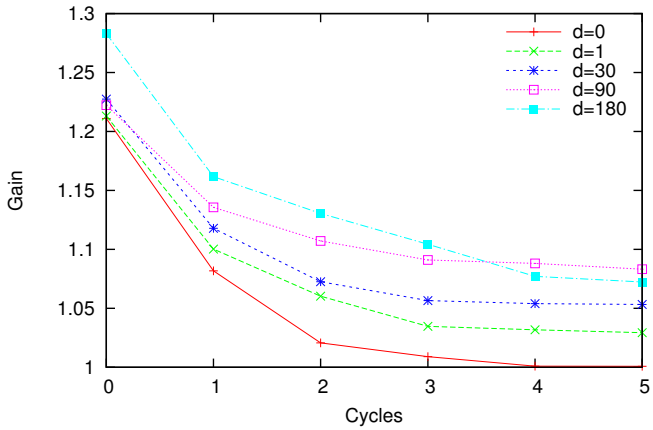


Figure 3.21: Impact of interest change

the result quality.

We mimic the change of tagging profiles as follows: giving a time period of d days, for a query issued by a user during this period, if the desired item for this query is found in the top-10 results, the tagging actions related to this item are added to the profile. The underlying intuition is that when a user discover a new item through querying, she is likely to tag this item by the tags used in the query to describe it. The larger d, the more tagging actions are added to the profile and the more the changes.

Figure 3.21 depicts the impact of interest change on the result quality. When the users' profiles change, the result quality based on cache is always better than the case where no cache is used ($Gain > 1$). This is due to the fact that when new tagging actions that do not use the tags in the query are added to the profile, the importance of the query tags in the profile vector decreases. The offer network is persistently maintained so that the neighbors in it become less correlated to the query tags. In contrast, the demand networks in the cache are selected if they give more emphasis on the query tags. Encouragingly, the more the users change their profiles, the better the result quality benefits from the cache. This conveys the effectiveness of using cache in DT^{2}P^2 to boost the query processing with on-line personalization.

3.3.5.3 Cost

Storage requirements As presented above, in our system, each user stores *(i)* profile vectors and entire profiles for the neighbors in her offer network and *(ii)* profile vectors for the neighbors in the demand networks in her cache. We use an objective metric, similar to that in [12], to measure the space requirement for each user. For a profile vector, each element can be presented as an entry in the form of $\langle tag, weight \rangle$ and the required storage can be estimated by the total number of entries it contains. A profile is a list of tagging actions. Similarly, the space required for storing a profile can be estimated by the number of tagging actions it contains.

Figure 3.22 shows the storage requirement for each user. The total length of the profiles in the offer network, the total length of the profile vector in the offer network and the cache are ranked

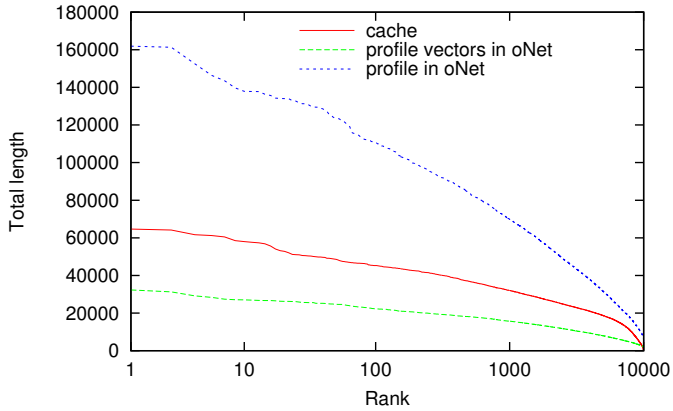


Figure 3.22: Storage requirement

respectively. We observe that the required space for a user mainly depends on the profiles in her offer network, which is about twice the space for maintaining the cache. Using the cache improves the query processing without imposing much overhead in storage.

Bandwidth Due to the periodical maintenance of the offer network and the burst communication for refining the demand network and processing the query, data are continuously exchanged in the system. As the maintenance of offer network is quite similar to that in the lazy mode of P4Q, we focus here on the bandwidth consumption due to the query processing and emphasize the impact of cache on it.

Two kinds of information are transmitted during the query processing: *(i)* the profile vectors for refining the querier's demand network and *(ii)* the partial result lists of each new neighbors in the demand network for processing the query. Each element in the profile vector can be presented as an entry in the form of $\langle tag, weight \rangle$. The bandwidth due to the transmission of the profile vectors thus mainly depends on the total length of these vectors. Similarly, each partial result list is composed of entries in the form of $\langle item, score \rangle$. Considering that the size of an item is application specific, we also use the total length of these partial result lists as a measure of the bandwidth consumption for transmitting them.

Figure 3.23 illustrates the total number of entries received by each querier when processing her query if no cache is used. As the querier stores the profiles of the neighbors in her offer network, no transmission is required to process the query in her initial demand network. The total length of the partial result lists decreases quickly and this explains why the improvement of query results mainly occurs in the first few cycles. The transmission of profile vectors remains relatively stable because at each cycle the same number of such vectors are transmitted until the query is solved.

Figure 3.24 shows how the bandwidth varies if the cache is used by each user. As we observed from Figure 3.23, the profile vectors transmitted at each cycle is relatively stable, only the transmission of the partial result lists is compared in Figure 3.24. When the cache is used, the transmission is mainly due to the query processing within the initial demand network (Cycle 0) because the querier needs to contact certain neighbors in this network to fetch the partial result lists derived from their

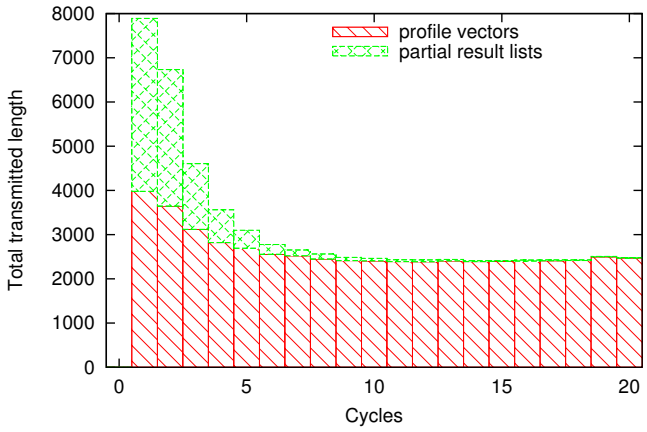


Figure 3.23: Transmission without cache

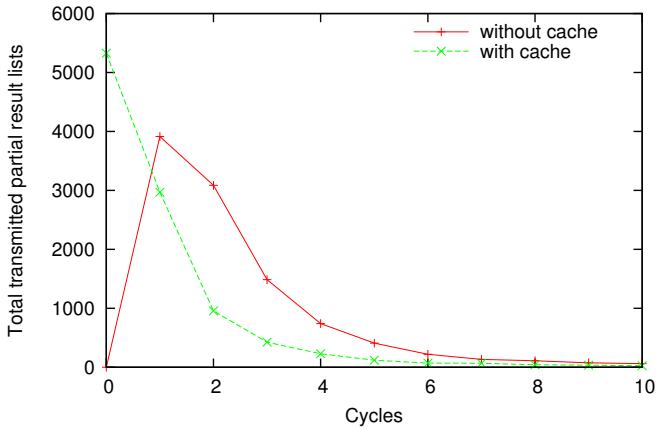


Figure 3.24: Transmission of partial result lists

profiles. In the following cycles, less information is transmitted when the cache is used. In fact, the total quantity of transmission is similar in both cases. Using the cache allows discovering the candidate items earlier and thus improving the result quality faster.

3.3.5.4 Summary

Our evaluation demonstrates that on-line personalized query processing can be efficiently achieved if the queriers virtually move in the system through gossiping with their neighbors. The same results as those obtained in a centralized system (DT^2) with global information can be obtained within 15 gossip cycles. If the cache, containing at most 5 cached demand networks, is used, the top-10 results improve more than 20% for the first few cycles. If the query corresponds to the emerging interests of the querier, this improvement can be up to 2.8 times better. The cache also guarantees

better results (10% per cycle) when the users interests change. There is a trade-off between the query response time and the bandwidth consumption as the quantity of data transmitted in each cycle is fixed (Figure 3.23). The shorter a gossip cycle, the more data is need to be transmitted for each time unit. Assuming that the querier gossips every 1 second for refining her demand network, which gives the querier enough time to assess the intermediate results, 15 seconds would be enough to answer a query. This in turn requires on average a bandwidth of 63 Kbytes per second. More adequate bandwidth would surely accelerate the query processing.

3.3.6 Concluding remarks

Personalized top-k processing in collaborative tagging systems has attracted a lot of research interests in recent years. Both centralized [12, 66, 103] and decentralized [117, 118, 119] approaches have been proposed to improve the effectiveness of the query processing in such environment. All of these approaches rely on implicit affinities among users to personalize the query processing. Yet, only the past tagging behaviors of the users are actually considered, which makes it difficult to fulfill the emerging interests of the queriers.

The on-line personalization approach DT^2 proposed in Section 3.2 addresses this problem and copes well with the queries reflecting emerging interests. Query processing is personalized by associating each querier with a set of social acquaintances whose tagging profiles are correlated to the querier's hybrid interest reflected in both her past tagging behavior and the query. This on-line personalization is achieved in a centralized system by doing top-k twice.

The DT^{2}P^2 protocol we proposed in this section relies on gossip to discover and leverage the implicit relations defined by the hybrid interest to provide the on-line personalized top-k processing in peer-to-peer systems. As we have shown in the evaluation, this approach is very efficient to perform the query processing and guarantees the same result quality as DT^2.

We also use a cache of demand networks to accelerate the query processing. Caching is a useful technique for Web search engines that are accessed by a large number of users. It enables shorter average response time and reduces the workload on the search engines. In general, there are two possible ways to use a cache: caching the answers [120, 121] or caching the inverted lists [122, 123]. The advantages and disadvantages of both ways are surveyed in [124]. The way we use the cache is inspired from the inverted lists caching but different from it. Caching the demand networks is more flexible in the sense that it does not aim to guarantee the results but only provides some access points that shorten the path from the querier to the good results. The efficiency of this demand network cache is conveyed by our evaluation.

DT^{2}P^2 achieves on-line personalization through gossiping in a fully decentralized collaborative tagging system. Yet, we believe there is room to improve its performance. For instance, instead of fetching all the profiles that contain at least one tags of the query, the querier can forward both the query and her profile vector to the user gossiping with her. In this way, this user can pre-compute the qualification of the neighbors in her offer network and only send those who are likely to be neighbors in the querier's demand network. This is expected to reduce the bandwidth consumption during the gossip. In addition, the querier has no need to fetch the entire partial result lists. The algorithms of TPUT family [73, 74, 75, 76] can be easily extended to this context for saving the bandwidth during the query processing.

We focus in this work on how to efficiently achieve the same on-line personalization as in a centralized system. It would be interesting to compare DT^{2}P^2 with other approaches [82, 96, 99]

that also process top-k queries in peer-to-peer systems. These approaches differ from DT^{2}P^2 as they all retrieve relevant information through query routing, which may lead to significant difference in term of query response time and bandwidth consumption.

3.4 Conclusion

We presented in this chapter two algorithms, DT^2 and DT^{2}P^2, that provide personalized top-k processing on-line in centralized and peer-to-peer collaborative tagging systems respectively. We described the design and the implementation of the two algorithms and investigated their performances using real datasets. The experimental results assess that on-line personalization is promising to fit the diversity of user preferences as well as their emerging interests. Performing the on-line personalization by doing top-k twice in a centralized system allows to narrow down the query processing while decentralized solution allows this on-line personalization to go further in a more scalable manner.

CHAPTER 4

CONCLUSION

4.1 Contributions

The Web 2.0 revolution has transformed the Internet from a passive read-only infrastructure to an active read-write platform. The content of the Web 2.0 applications, especially that of the collaborative tagging systems, is generated by every participant and is annotated with their freely chosen tags. As a result of the rapidly growing quantity of such content, the user generated taxonomy, aka folksonomy, has exposed its huge potential for search content in collaborative tagging systems. But such a search can turn into a nightmare since the user freedom to choose tags reveals a big source of ambiguities.

Personalization is an appealing way to disambiguate the search, as well as fulfill the diverse user preferences hidden behind the queries. This is achieved in this thesis by exploiting the information from the social acquaintances sharing similar interests with each querier.

We first proposed the protocol P3K, which decentralizes a state-of-the-art approach [12] and achieves off-line personalized top-k processing in peer-to-peer systems. A gossip-based protocol is employed to capture the implicit relationships among users and organize similar users in query-efficient personal networks. Relevant information is locally maintained by each user to enable efficient processing for her own queries. A personal network limiting procedure is also proposed. P3K alleviates the scalability problem faced by the centralized solution, while preserving almost the same result quality.

The second protocol we proposed, P4Q, goes one step further and enhances the system performance in terms of storage, bandwidth and dynamics adaptability. P4Q relies on a bimodal gossip to personalize the query processing. The maintenance of personal networks is performed in a lazy mode, with a fairly low frequency to avoid overloading the system. The query processing is performed in an eager mode, with increased frequency to reduce the latency of the query processing. Queries are gossiped among social acquaintances and processed collaboratively on-the-fly. To limit the bandwidth consumption, the gossip of profiles is decomposed in 3 steps, relying on a similarity estimation mechanism based on profile digests encoded in Bloom filters. A response mechanism,

which 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, is also incorporated in P4Q. P4Q provides efficient off-line personalized query processing in fully decentralized systems with enhanced performance.

Since users may issue queries that reflect their emerging interests and are difficult to satisfy with the off-line personalization, we proposed an on-line personalization scheme that relies on a hybrid interest model to improve the search.

To apply the proposed on-line personalization scheme in centralized systems, the DT^2 algorithm was proposed. DT^2 performs personalized query processing at query time by doing top-k twice. The first top-k processing relies on a threshold algorithm to determine the personal network of the querier according to her hybrid interest. The second top-k processing screens the items possessed by the top-k users to obtain the query results. The users are folded into the personal networks in an incremental manner to guarantee the balance between the result quality and the search efficiency. In fact, DT^2 provides high quality results for both ordinary and emerging interests and narrows down the processing in terms of storage and time.

To apply the same on-line personalization scheme in peer-to-peer systems, we further proposed the DT^2P^2 algorithm. DT^2P^2 relies on a gossip-protocol to maintain a multi-objective overlay, where the offer network of each user associates her with social acquaintances sharing sustainable interests and the demand network defined by her hybrid interest serves the query processing. Queries are iteratively processed within the demand network at each gossip cycle. A cache of demand networks is used to accelerate the query processing. DT^2P^2 efficiently provides the same result quality as DT^2 without any global information. It also exhibits good performance in the face of user departing and interest change.

To summarize, the contribution of this thesis lies in *(i)* the realization of the off-line personalized query processing in large-scale peer-to-peer systems; *(ii)* the proposition of the on-line personalization query processing as well as a centralized realization; *(iii)* the realization of the proposed on-line personalization in large-scale peer-to-peer systems. It is interesting to notice that, although the proposed P4Q and DT^2P^2 algorithms aim to provide different personalization, be they off-line or on-line, the underlying mechanisms are similar. Better performance might be expected through their combination: queries reflecting ordinary interests of the querier could be processed by gossiping the query while queries reflecting emerging interests could be processed by building a demand network. This is potentially part of the future work.

4.2 Perspectives

We discuss in this section further issues that we believe to be interesting to explore. We foresee our future research along two axes: anonymous query processing and persistent query processing.

Anonymity in personalized peer-to-peer query processing Monitoring and storing any kind of user activity is a sensitive issue. Although collaborative tagging sites, as well as the most of Web 2.0 applications, are collaborative by nature, some users may not want to see their interests exposed. In centralized approaches, the tagging behavior of every user is known to the central server but no explicit association between a user and her interest is available to other users. A decentralized approach prevents the danger of a central authority making commercial usage of the profiles of all users, but might disclose the profiles of users to other users. Indeed, the proposed protocols (P3K,

P4Q and DT^{2}P^2) rely on users exchanging profiles in a peer-to-peer manner to maintain the personal network and processing the queries.

Interestingly, in P3K although the personalized inverted lists of each user depend on her personal network, users do not need to explicitly know from whom this information comes. The anonymity can be preserved by decoupling the user identity (e.g. contact information like IP address) and the profile during the gossip. Besides the one discussed in [125], there are two possibilities to achieve such decoupling:

- Full decoupling: Each user only maintains a set of random user identities and a set of random profiles for her random view. At each gossip, a user gossips with a random neighbor and exchanges random profiles with her. The personal network is fed by the appropriate profiles but there is no indication whose profiles they are. This strategy can guarantee the anonymity but may take more time for each user to build her personal network.

- Partial decoupling: Each user maintains a set of random user identities in her random view and the right profiles in her personal network. At each gossip, a subset of profiles in the personal network is exchanged with a random neighbor. This strategy may improve the efficiency of building the personal network but it would decrease the anonymity. A user who receives the profiles can infer the interests of the user gossiping with her since the received users share similar interests with that user.

In practice, not all the people care about anonymity. In contrast, some users might even consider sharing their interests as a way to make new friends. It is thus interesting to investigate the impact of these two possibilities on the anonymity and the efficiency of the query processing in P3K.

However, for P4Q and DT^{2}P^2, queries are collaboratively processed by several users. Receiving directly the partial results contributed by these users would expose their interests to each other. To address this problem, one can think of transmitting the query and the partial results via some third part users randomly selected from the system. Some encryption strategies need be considered to make sure that the third part users are not able to associate the user identity with the query or the partial results. This represents an interesting perspective that can be explored.

Persistent personalized peer-to-peer query processing The protocols proposed in this thesis aim to provide instant query processing, i.e., each query is processed only once with the information available in the system at query time. Yet, in Web 2.0 applications, as well as collaborative tagging systems, new contents are created and shared at a very high pace. As a consequence, users may be willing to keep receiving the most recent results related to their queries. The '*following*' operation in *Twitter* is a nice attempt for users to keep up with the most up-to-date information they are interested in.

Persistent search allows users to issue the query once and receive real-time updates whenever there is a new result for that query. This ensures that users are always on the pulse of what is going on, while liberating the users from repeating the query themselves.

Publish/Subscribe systems [126] are designed to disseminate messages in peer-to-peer environments, from users (peers) issuing events (publisher) to users interested in those events (subscriber). Subscribers typically register with the system the type of events they are interested in by submitting a persistent query for instance. Publishers simply post events. The system is responsible for

delivering the right events to the right users. These systems are very effective to deal with persistent queries. Yet, with the personalization requirement, it would become difficult for the system to maintain fine-grained information about the preferences of each user and process the queries efficiently.

In DT^2P^2, the personalized queries are processed on-line. A querier virtually "moves" in the peer-to-peer system by refining her demand network, which allows her to approach the desired results gradually. It is natural to extend this idea to persistent query processing. In other words, different from Publish/Subscribe systems, where the queriers passively wait for incoming messages, they can move, like DT^2P^2 users, in the network to actively discover the results. The only difference lies in the fact that the demand network of a query is persistently maintained as long as the querier does not abolish that query. Putting this to work is however challenging. Several critical issues should be considered:

- Similarity measure: An adaptive metric that measures the similarity between users should be defined. Since the querier's demand network is refined based on this metric, it is important to define and dynamically adjust this metric in such a way that the most recent changes in the system (that match the query) could always be involved in the resulting demand network. To this end, in addition to the profile and the query, many other factors might need to be taken in to account, including the quality of the current query results, the arrival rate of the new results, the activity level of the neighbors, etc.

- Cost issue: A user may issue several persistent queries during a same period. Maintaining a demand network for each query might be costly in terms of bandwidth and storage. Therefore, it would be interesting to investigate some mechanisms that exploit the similarity among queries and refine their demand networks in a more efficient way to reduce the unnecessary cost.

Despite these challenges, we believe that persistent personalized query processing in fully decentralized systems is a promising way to efficiently improve the user satisfaction in nowadays highly dynamic and prolific networks, and it is worth exploring.

APPENDIX A

RÉSUMÉ ÉTENDU

Les réseaux sociaux, les blogs, les wikis, les sites de partage de contenus sont toutes des facettes du Web 2.0 qui dépendent des utilisateurs dans le service qui rendent. Dans cette nouvelle approche du Web, les utilisateurs ne sont plus uniquement des lecteurs passifs mais ils peuvent aussi donner leur avis sur les contenus du Web, interagir mais aussi fournir le contenu. Le contenu généré par les utilisateurs constitue une proportion considérable du Web et devient une source d'informations de plus en plus riche.

Une des applications émergentes du Web 2.0 est l'annotation collaborative de contenu qui permet aux utilisateurs de donner leur avis sur le contenu du Web en associant aux éléments de contenu des mots clés, appelé *tag*. Cela permet aux utilisateurs de créer et de gérer librement des tags pour annoter le contenu, qui aide les utilisateurs à se souvenir et organiser les informations, comme le courrier électronique (*Gmail*), sites web (*delicious*), photos (*Flickr*), vidéos (*Youtube*), blogs (*Technorati*) et des documents académiques (*CiteULike*). Ces tags peuvent ensuite être utilisés par des utilisateurs pour rechercher cette information.

Une caractéristique essentielle de l'annotation collaborative est de favoriser la navigation sociale. Les utilisateurs n'ont pas de contrainte sur les mots clés qu'ils utilisent pour décrire une ressource. Ils peuvent même inventer leurs propres tags ou leur associer une signification personnelle pour faciliter l'organisation de leurs ressources. Selon l'analyse de [14], seulement 50% des URLs annotés dans *delicious* contiennent dans leur page les tags utilisés pour les annoter. Ceci révèle que l'utilisation des tags générés par les utilisateurs est un bon complément à l'indexation du texte des pages Web. En ce qui concerne les photos et les vidéos qui ne contiennent pas de texte, les tags générés par les utilisateurs fournissent une description exploitable par la procédure de recherche. Ceci aurait un impact important sur l'amélioration de la qualité de la recherche.

Pourtant, effectuer efficacement des recherches en utilisant les tags générés par les utilisateurs est difficile, en particulier lorsque ces recherches engendrent des ambiguïtés. Par exemple, si un informaticien recherche "*matrix*" dans Google, il est probablement à la recherche de notions mathématiques. Cependant, les premières pages fournies par *Google* sont toutes sur le film "*Matrix*". En revanche, un fan de *Keanu Reeves* pourrait être à la recherche de ce film. La nature non structurée

des mots clés, l'absence d'une ontologie fixe et la diversité des préférences des utilisateurs rendent la recherche plus difficile parce que la valeur d'une même information peut varier d'un utilisateur à l'autre. Par exemple, le même tutoriel pour "*Latex*" peut être très intéressant pour un expert qui recherche des améliorations spécifiques, mais difficile pour un débutant à cause de l'absence d'exemples intuitifs.

La personnalisation est une méthode prometteuse dans ce contexte pour éviter ces écueils tout en limitant l'espace de recherche à un sous-ensemble d'informations pertinentes, permettant de lever l'ambiguïté sur le traitement des requêtes. Nous effectuons donc nos travaux dans le cadre des systèmes collaboratifs et nous présentons dans cette thèse, comment effectuer une recherche efficace dans de tels systèmes en personnalisant le traitement des requêtes avec les tags générés par les utilisateurs.

Objectifs et positionnement

Plusieurs approches personnalisées ont été proposées pour exploiter les réseaux sociaux figurant dans des systèmes collaboratifs au sein des procédures de recherche [65, 99]. Jusqu'à présent, ces approches ont porté principalement sur les réseaux sociaux explicites, établis a priori, indépendamment des profils de l'utilisateur (par exemple *Facebook*). Nous plaidons pour l'amélioration de la qualité de la recherche d'information en exploitant la corrélation implicite entre les utilisateurs ayant des intérêts similaires. La motivation vient de l'observation que des personnes inconnues, mais avec qui nous partageons de nombreux intérêts, peuvent être très utile pour rechercher sur le Web.

Des approches [67, 60, 72] très élégantes ont récemment été proposées pour exploiter cette personnalisation implicite par des modèles sociaux. Diverses notions d'affinités ont également été développées pour explorer les similarités entre les utilisateurs [63, 64, 66, 113]. Ces travaux précédents confirment que les relations implicites entre les utilisateurs sont très efficaces pour améliorer la qualité des recherches. Pourtant, ces travaux s'appuient essentiellement sur le comportement passé des utilisateurs pour en déduire leurs préférences, tels que leur historique d'annotation [67, 63], leur historique de recherche [64, 72] ou leur historique de navigation [60]. Il est donc difficile de satisfaire les intérêts émergents des utilisateurs. Bien que la notion de l'histoire récente a été proposée dans [72] et [114], ces approches permettent seulement d'obtenir de bons résultats lorsque les requêtes sont corrélées aux événements récents. Nous proposons une nouvelle méthode de personnalisation qui est appropriée pour toutes sortes de requêtes.

Les solutions centralisées posent naturellement des problèmes de passage à l'échelle. Premièrement, le maintien des informations précises sur les utilisateurs est extrêmement consommatrice d'espace. En vertu de l'estimation de [12], plusieurs téraoctets sont nécessaires pour permettre de personnaliser le traitement de requêtes top-k dans un système comme *delicious* contenant seulement 100,000 utilisateurs. Considérant que *delicious* a actuellement des millions d'utilisateurs, un système de recherche centralisé de personnalisation semble impossible à mettre en place. De plus les utilisateurs des systèmes collaboratifs sont très actifs; ils changent leur profil en effectuant constamment de nouvelles annotations sur de nouveaux contenus. Il est difficile pour un système centralisé de prendre en compte en temps réel ces changements.

Nous soutenons qu'un système de recherche dans les réseaux sociaux implicites appelle à des solutions décentralisées. En plus d'être capable de passer à l'échelle et de faire face à des changements constants des profils des utilisateurs, les solutions décentralisées évitent le danger des autorités cen-

trales qui pourraient abuser des informations à leur disposition. Les limites des systèmes centralisés ont été illustrées par l'exploitation de profils d'utilisateurs à des fins commerciales, ou par les pannes de déni de service *Facebook*, *Twitter* et *LiveJournal* suite à des attaques en août 2009 par exemple. Par conséquent, nous visons à assurer le traitement à large échelle des requêtes personnalisées de façon efficace pour des réseaux sociaux implicites en utilisant le paradigme pair-à-pair.

A notre connaissance, seuls quelques travaux abordent le problème de personnalisation dans des systèmes décentralisés. Malheureusement, ces approches ne sont pas assez satisfaisantes car soit elles imposent des contraintes sur les activités de l'utilisateur [98] ou elles exigent des liens supplémentaires entre les données [97]. Nous nous concentrons principalement sur la façon d'identifier et d'exploiter les aspects sociaux dans le traitement des requêtes de top-k de façon à améliorer la qualité des résultats en pénalisant le moins possible les utilisateurs peu actifs vis-à-vis des utilisateurs qui participent beaucoup au système. Plus important encore, contrairement aux travaux cités précédemment, nous allons concevoir des algorithmes spécifiques aux systèmes collaboratifs.

Plan de la thèse

Dans cette thèse nous visons à fournir une recherche efficace pour les applications du Web 2.0. Nous mettons en évidence la possibilité d'utiliser les tags générés par les utilisateurs pour personnaliser la recherche et l'applicabilité de la recherche personnalisée dans les environnements entièrement décentralisés. Dans ce travail, nous nous concentrons sur les systèmes collaboratifs, où les utilisateurs ajoutent des métadonnées sous la forme de tag pour décrire et partager le contenu. Plus précisément, nous considérons le comportement des utilisateurs comme indicateur de leurs préférences et associons à chaque utilisateur un ensemble d'autres utilisateurs partageant des intérêts similaires afin de personnaliser le traitement des requêtes top-k. Les utilisateurs similaires peuvent être identifiés, soit hors ligne ou en ligne selon des exigences de personnalisation.

Cette thèse propose de nouveaux algorithmes permettant d'effectuer des recherches personnalisées de manière efficace dans des systèmes dynamiques, centralisés ou décentralisés, selon deux axes majeurs : (i) la personnalisation hors ligne qui s'appuie sur le comportement passé des utilisateurs et (ii) la personnalisation en ligne qui s'appuie sur le comportement passé et la requête en cours. Par conséquent, cette thèse est organisée en deux chapitres principaux (chapitre 2 et chapitre 3). Ces deux chapitres sont respectivement dédiés à la personnalisation du traitement des requêtes hors ligne et en ligne.

Dans le chapitre 2, nous présentons d'abord l'algorithme P3K, qui décentralise une approche existante et réalise le traitement personnalisé des requêtes top-k hors ligne dans les systèmes pair-à-pair. Ensuite, nous présentons P4Q, une extension de P3K qui améliore les performances du système en termes de stockage, bande passante et la robustesse en distribuant le traitement des requêtes.

Dans le chapitre 3, afin d'améliorer encore la qualité des résultats pour les requêtes représentant les intérêts émergents des utilisateurs, et donc non représentés dans son profil, nous proposons un modèle hybride d'intérêt, prenant en compte à la fois le profil des utilisateurs mais également la requête elle-même. Nous proposons une solution à la fois en centralisé, l'algorithme DT^2, qui effectue une recherche de type top-k à deux reprises. L'algorithme DT^2P^2 exécute efficacement la personnalisation en ligne de manière entièrement décentralisée.

Avant les deux chapitres principaux, dans le chapitre 1, nous introduisons d'abord les systèmes collaboratifs et les systèmes pair-à-pair à grande-échelle dans lesquels nos recherches sont menées. Ensuite, nous présentons les principes fondamentaux des algorithmes épidémiques et des algo-

rithmes du traitement des requêtes top-k, utilisés respectivement pour la construction des réseaux sociaux et le traitement des requêtes top-k. Nous finissons ce chapitre en présentant des approches existantes qui personnalisent le traitement des requêtes top-k dans les systèmes centralisés ou décentralisés, puis discutons comment notre travail diffère de ces travaux précédents.

Enfin, dans le chapitre 4, nous concluons en réfléchissant aux améliorations futures des algorithmes décentralisés proposés en terme de l'anonymat au sein de la personnalisation. Nous présentons également de nouvelles fonctionnalités, comme le traitement des requêtes persistantes, qui pourraient être intégrées à l'algorithme DT^{2}P^2 pour le rendre plus riche. Nous décrivons quelques critères qui permettent de mesurer des similarités entre utilisateurs en temps réel et satisfaire les besoins des utilisateurs d'une manière continue et efficace.

Contributions

Les contributions de cette thèse sont récapitulées par le tableau A.1. Elles se décomposent en deux axes, personnalisation hors ligne et personnalisation en ligne, correspondant aux deux chapitres principales de cette thèse. Parmi les algorithmes proposés, P3K et P4Q réalisent la personalisation hors ligne dans les système décentralisés et DT^2 et DT^{2}P^2 réalisent la personnalisation en ligne dans les système centralisés et décentralisés respectivement. Nous détaillons les différentes contributions de cette thèse selon les deux axes dans la suite.

Table A.1: Positionnement des contributions

	Personnalisation hors ligne	Personnalisation en ligne
Centralisé		DT^2
Décentralisé	P3K, P4Q	DT^{2}P^2

Traitement des requêtes hors ligne

Le traitement personnalisé hors ligne des requête top-k a été proposée, pour la première fois, dans [12] pour les systèmes collaboratifs centralisés. L'idée est de construire et de maintenir pour chaque utilisateur, hors ligne, un réseau personnel des utilisateurs ayant des comportements d'annotation similaires pour traiter sa requête. Plus précisément, étant donné une requête $Q = \{t_1, ..., t_n\}$, émise par un utilisateur u_i avec un ensemble de tags $t_1, ..., t_n$, le traitement personnalisé de la requête Q vise à retourner un ensemble d'objets ayant les scores les plus élevés dans le réseau personnel de u_i. La personnalisation consiste à limiter l'espace de recherche à un sous-ensemble du système, c'est à dire des utilisateurs ayant des intérêts similaires avec l'interrogateur. Nous appelons cela une personnalisation hors ligne car le réseau personnel de chaque utilisateur est construit indépendamment du traitement des requêtes. En d'autres termes, le réseau personnel de chaque utilisateur est pré-calculé et connu au moment de la requête. Une fois qu'un utilisateur émet une requête, le système n'a besoin que de traiter cette requête au sein de son réseau personnel pour obtenir les résultats personnalisés.

Toutefois, les solutions centralisées pour mettre en œuvre cette personnalisation s'avèrent difficiles compte tenu du volume important d'informations qui doit être maintenu pour chaque utilisateur. Par exemple, la stratégie la plus simple dans [12], dite *Exact*, consiste à maintenir une liste inversée par paire $\langle utilisateur, tag \rangle$. Le score d'un objet dans une liste inversée $\langle u_i, t_n \rangle$ dépend de la façon

dont cet objet a été annoté par les utilisateurs dans le réseau personnel de u_i. Par conséquent, si un tag est utilisé dans les réseaux personnels de tous les utilisateurs, autant de listes inversées devront être maintenues. Ceci nécessite énormément d'espace en raison des informations massivement dupliquées. Des solutions alternatives pour économiser l'espace de stockage sont possibles, mais le temps de traitement est augmenté et ce n'est pas très souple quand les utilisateurs sont dynamiques.

L'algorithme P3K Pour tenir compte de ces défis, nous proposons d'abord l'algorithme P3K, qui décentralise l'approche *Exact* dans [12] et réalise le traitement personnalisé des requêtes top-k hors ligne dans les systèmes pair-à-pair. Dans P3K, chaque utilisateur découvre et maintient périodiquement un ensemble de voisins qui forment son réseau personnel. Les informations contenues dans le profil des voisins dans le réseau personnel d'un utilisateur sont organisées dans les listes inversées pour chaque paire $\langle utilisateur, tag\rangle$ et stockés par elle-même. Chaque utilisateur traite localement ses propres requêtes avec les listes inversées stockées. Ce contrôle total du réseau personnel permet une personnalisation complète du traitement de la requête. Plus important encore, la capacité de traitement et de stockage augmente d'une façon linéaire avec le nombre d'utilisateurs en évitant les goulets d'étranglement potentiels inhérent à un seul serveur.

Un protocole épidémique est utilisé pour capturer les relations implicites entre les utilisateurs et organiser les utilisateurs similaires dans le réseau personnel de chaque utilisateur d'une manière dynamique. À chaque période, chaque utilisateur gossipe avec un de ses voisins. Elles échangent leurs voisins dans leurs réseaux personnels. Chaque utilisateur sélectionne alors les utilisateurs les plus similaires pour mettre à jour son réseau personnel. Une fois que ces réseaux personnels sont mis en place, les utilisateurs recueillent les informations pertinentes de leurs voisins localement et traitent leurs requêtes en utilisant un algorithme classique de top-k, dénommé NRA [46, 52]. Sans contrainte de taille sur le réseau personnel, il peut croître indéfiniment. Cela peut générer un grand nombre d'échanges d'informations et augmenter les besoins de stockage. Dans la pratique, un sous-ensemble des utilisateurs soigneusement sélectionnés est suffisant pour fournir la plupart des informations utiles pour le traitement des requêtes top-k. Nous limitons donc le réseau personnel de chaque utilisateur à une certaine taille de sorte que seuls les utilisateurs avec les plus grandes similarités soient impliqués dans le réseau personnel d'un utilisateur donné. À cet effet, une nouvelle mesure est proposée pour affiner les similarités des voisins dans le réseau personnel illimité et faire apparaître les plus pertinents qui forment le réseau personnel de P3K.

Nous évaluons P3K, à l'aide de *PeerSim* [105] avec un ensemble de données *delicious* de 10,000 utilisateurs collectés en janvier 2009. Les résultats expérimentaux montrent que si tous les utilisateurs qualifiés sont conservés dans le réseau personnel, après 50 cycles de gossips, on retrouve au moins 8 objets pertinents sur 10 (figure 2.7) avec un stockage négligeable par rapport à la solution idéale centralisée *Exact* [12] (figure 2.10). Grâce à la stratégie d'optimisation de P3K qui ne garde que les voisins les plus proches dans le réseau personnel, plus de 30% de stockage est en outre économisé pour chaque utilisateur (figure 2.11). En fait, P3K permet de récupérer presque les mêmes résultats (figure 2.8) avec une légère pénalité dans le temps de traitement (figure 2.13), par rapport à la solution la plus rapide parmi celles de [12]. Nous montrons également que P3K passe à l'échelle : la taille nécessaire du réseau personnel pour obtenir les résultats de la même qualité reste stable même avec un nombre croissant d'utilisateurs dans le système (figure 2.14). Pour résumer, les résultats expérimentaux de P3K confirment que la décentralisation est une voie prometteuse pour assurer que le traitement personnalisé des requêtes top-k passe à l'échelle.

L'algorithme P4Q Ensuite, nous proposons P4Q, une extension de P3K qui améliore les performances du système en termes de stockage, bande passante et la robustesse en distribuant le traitement des requêtes.

Comme P3K, P4Q ne repose pas sur un serveur central. En revanche, P4Q est une solution basée sur un protocole de gossip bimodal pour personnaliser le traitement des requêtes dans des systèmes pair-à-pair. Le mode *paresseux* s'exécute périodiquement à une fréquence faible afin de maintenir le réseau personnel. Le mode *agressif* fonctionne à la demande et est en charge du traitement collaboratif des requêtes tout en rafraîchissant une partie spécifique de réseaux personnels des utilisateurs. Le mode *agressif* est activé uniquement lors de la requête et s'arrête lorsque la requête est précisément calculée.

Le maintien du réseau personnel est effectué avec le mode *paresseux*, à une fréquence assez faible pour éviter de surcharger le réseau. Les utilisateurs maintiennent régulièrement leurs réseaux personnels en gossipant entre eux et en calculant les similarités entre les profils comme dans P3K. Pourtant, chaque utilisateur stocke seulement localement un nombre limité de profils, notamment ceux de ses voisins les plus similaires. Le nombre de profils stockés sur chaque utilisateur est choisi en fonction de sa capacité de stockage.

Pour limiter la consommation de bande passante, les utilisateurs n'échangent les profils de leurs voisins durant le gossip que lorsque ceux-ci paraissent très similaires à leur propre profil. Ainsi, des résumés de profils, codés dans les filtres de Bloom [108], sont d'abord échangés pour estimer une limite supérieure de la similarité entre deux profils. Il est possible pour un utilisateur u_j de devenir un voisin de l'utilisateur u_i si cette estimation de la limite supérieure est plus élevée que la similarité entre u_i et le voisin le moins similaire dans son réseau personnel, ou cette limite supérieure est supérieure à 0 tant que l'utilisateur n'a pas encore le nombre désiré de voisins dans son réseau personnel. Les actions d'annotation qui contribuent au calcul de la limite supérieure sont transmises pour calculer la similarité exacte. Comme seuls les profils d'un sous-ensemble d'utilisateurs ayant les similarités les plus importantes seront stockés, cela permet de réduire encore la consommation de bande passante en évitant la transmission des profils inutiles. Pour assurer une bonne précision de cette estimation, le filtre de Bloom est basé à la fois sur les tags et les objets contenus dans le profil d'un utilisateur. La taille du filtre de Bloom est dynamiquement adaptée à la taille du profil codé afin de garantir un faible taux de faux positifs[1].

Le traitement des requêtes est basé sur le mode *agressif* de P4Q, c'est à dire avec une fréquence accrue et est orientée vers les relations sociales. Chaque requête est d'abord calculée localement, sur la base de l'ensemble des profils stockés, fournissant un résultat immédiat et partiel à l'utilisateur. Ensuite, la requête, ainsi que la liste des profils nécessaires pour la traiter, est gossipé et calculé d'une manière collaborative. La requête est gossipée d'abord au plus proche voisin et ensuite plus loin en fonction de la proximité sociale entre les profils d'utilisateurs, d'une façon itérative pour affiner les résultats. Chaque utilisateur atteint par la requête calcule localement sa part de la requête sur la base des profils pertinents stockés localement, puis la gossipe plus loin. Les résultats sont donc itérativement affinés dans un certain nombre de cycles de gossip, en récoltant des informations pertinentes à chaque étape, et affichés directement à l'interrogateur. Comme le nombre de résultats partiels à fusionner varie avec le temps, nous utilisons une variante de l'algorithme NRA pour

[1]Notez que les faux positifs ne conduirait pas à une inexactitude en sélectionnant les voisins pour le réseau personnel car une erreur de jugement ne surestime le score de la limite supérieure et la similarité exacte est calculée avant de transmettre le profil entier. Le seul impact des faux positifs est de transmettre certaines informations inutiles pour le calcul de la similarité exacte. Pourtant, un faible taux de faux positifs tient cet impact marginal.

récupérer le résultat top-k à partir des résultats partiels à chaque cycle. Cette variante garantit que chaque liste de résultat partiel est numérisé qu'une seule fois pendant l'ensemble du traitement. Une technique de partitionnement est utilisée pour éviter les utilisateurs de fausser les résultats en prenant en compte des informations redondantes. Gossiper la requête évite de saturer le réseau en communiquant avec tous les voisins dans le réseau personnel en même temps et rafraîchit la partie du réseau provenant de l'interrogateur, générant une onde spécifique de rafraîchissements dans le processus de personnalisation. L'utilisateur peut, à tout moment, consulter les résultats des requêtes et décider si ceux-ci sont assez satisfaisants.

Les utilisateurs de systèmes collaboratifs sont généralement actifs et annotent en permanence des nouveaux objets dans les systèmes. Pour garantir l'exactitude des résultats de la requête, P4Q contient un mécanisme qui prend en compte activement les modifications du profil de chaque utilisateur et garantit les rafraîchissements efficaces des réseaux personnels en termes de mise à jour des profils stockés et découverte de nouveaux voisins. Ceci est réalisé de manière collaborative par les utilisateurs participant au gossip à l'aide de deux opérations de base : *auto-promotion* et *aide-mutuelle*. *Auto-promotion* consiste, pour chaque utilisateur, à diffuser son profil lorsque celui-ci change, d'une manière proactive. *Aide-mutuelle* permet aux utilisateurs de partager avec d'autres utilisateurs leurs informations à jour.

Nous évaluons P4Q la fois analytiquement et expérimentalement. L'analyse montre que le temps de traitement d'une requête dans les cycles de gossip peut être approchée par $O(log_2 L)$, où L est le nombre de profils dans le réseaux personnel d'un utilisateur qui contribuent au traitement de la requête, mais ne sont pas stockées par l'utilisateur. En outre, l'analyse borne le nombre de messages engagés par la propagation de la requête et la transmission des résultats partiels. Nos évaluations expérimentales confirment les résultats d'analyse. Nous utilisons les mêmes données que celles de P3K. Nous considérons plusieurs scénarios de stockage. Nous montrons que même si chaque utilisateur stocke seulement 10 profils dans son réseau personnel, les requêtes top-k peuvent être exactement satisfaites en 10 cycles de gossip (figure 2.22), correspondant à 50 secondes avec un mode de fonctionnement *agressif* toutes les 5 secondes. Nous mettons en évidence le compromis entre les attentes de l'utilisateur sur les résultats de la requête, la latence de la réponse et la disponibilité du stockage. En exécutant la mode *paresseux* chaque minute, même si tous les utilisateurs modifient simultanément leurs profils, en une demi-heure, 95% des informations stockées sont mises à jour (figure 2.26) et les nouveaux voisins de plus de 50% des utilisateurs sont identifiés (figure 2.29). Le mode *agressif* génère aussi une onde de rafraichissements pour les utilisateurs qui participent au traitement des requêtes (figure 2.28). Ceci fournit aux utilisateurs une incitation à contribuer au traitement des requêtes des autres utilisateurs. Pendant ce temps, P4Q encourt une surcharge acceptable en termes de consommation de bande passante : 7,6 Kbits par seconde est suffisant pour maintenir le réseau personnel et 91 Kbits par seconde sont suffisants pour traiter une requête. P4Q est également résistant au départ des utilisateurs : un départ massif de 50% des utilisateurs diminue uniquement de 10% la qualité des résultats 10% (figure 2.30). Pour conclure, P4Q fournit un traitement personnalisé efficace des requêtes hors-ligne dans des systèmes entièrement décentralisés avec des performances améliorées.

Traitement des requêtes en ligne

Le approches de personnalisation hors ligne sont fondées sur l'hypothèse que les profils d'annotation sont suffisamment représentatifs des préférences de l'utilisateur. Ceci permet le dé-

couplage de la personnalisation et le traitement des requêtes en pré-calculant d'un sous-ensemble des utilisateurs hors ligne, ce qui garantit à son tour de diminuer la latence de réponse au moment de la requête. Ainsi, si une requête est fortement corrélée au profil de l'interrogateur, qui est utilisé pour modéliser ses préférences et sélectionner les voisins pour son réseau personnel, les résultats personnalisés sont plus satisfaisants que les résultats obtenus sans aucune spécification. Si la requête n'est pas corrélée au profil de l'interrogateur, représentant par exemple un interêt émergent, la qualité du résultat peut se dégrader. De toute évidence, afin de traiter efficacement les requêtes qui sont corrélées au profil de l'interrogateur, ainsi que ceux qui ne sont pas corrélées à ce profil, la personnalisation adéquate doit être effectuée sur la base de (i) le profil d'annotation de l'interrogateur et (ii) la requête elle-même. La mise en en pratique nécessite d'abord d'affiner les préférences des utilisateurs en utilisant à la fois le profil et la requête. Ceci exige l'exécution de la personnalisation en ligne parce que les intérêts émergents ne peuvent être pris en compte quand une requête est émise. À cet effet, nous étudions deux algorithmes pour effectuer un tel traitement personnalisé de la requête en ligne dans les systèmes centralisés et décentralisés.

L'algorithme DT^2 Nous proposons l'algorithme DT^2 pour effectuer en ligne le traitement personnalisé des requêtes top-k dans les systèmes collaboratifs centralisés. DT^2 effectue le traitement personnalisé des requêtes au moment de la requête en effectuant 2 top-k successifs.

Le premier protocole de top-k sous-jacent à DT^2 vise à déterminer un réseau de petite taille de k_1 utilisateurs appropriés pour traiter la requête, c'est à dire le réseau personnel d'un utilisateur pour une requête spécifique. Un aspect crucial ici est le calcul d'un *vecteur d'intérêt hybride* qui tient compte à la fois du profil d'annotation et de la requête elle-même, tout en attribuant dynamiquement le poids approprié à chacun. Deux aspects sont essentiels dans l'expression de cet intérêt hybride : le premier ajuste l'importance relative des tags dans la requête et le second ajuste l'importance de la requête par rapport au profil. Nous introduisons ensuite une métrique basée sur le calcul du cosinus de similarité hybride. Cette métrique est monotone, elle permet de comparer ce vecteur d'intérêt enter les profils des autres utilisateurs et de sélectionner les k_1 utilisateurs les plus appropriés en utilisant un algorithme classique TA [46].

Le deuxième protocole de top-k sous-jacent à DT^2 parcourt le réseau personnel formé par ces utilisateurs, et détermine les objets les plus appropriés. Nous montrons analytiquement que la construction de listes inversées, puis le traitement de la requête avec un algorithme traditionnel de top-k est plus coûteux que d'utiliser notre deuxième protocole de top-k.

L'originalité principale de DT^2 est dans l'exécution de la personnalisation en ligne d'abord en associant l'interrogateur d'un réseau personnel de k_1 utilisateurs, puis en traitant de la requête au sein de ce réseau pour trouver les k_2 objets les plus appropriés. Ainsi, choisir une valeur appropriée pour k_1 est cruciale pour la qualité des résultats : (i) une valeur trop faible de k_1 risque de faire exclure certains objets pertinents; (ii) une valeur trop grande de k_1 peut introduire des bruits inutiles en tenant compte des avis des utilisateurs avec des intérêts disjoints ou opposés et ainsi diluer l'effet positif de la personnalisation. En outre, plus la valeur de k_1 est grande, plus le temps nécessaire pour trouver ces utilisateurs est long. Il est donc important de parvenir à un équilibre entre la qualité des résultats et le temps de traitement. Dans DT^2, les utilisateurs sont considérés dans le réseau personnel d'un utilisateur d'une manière progressive. Si l'utilisateur ne semble pas satisfait avec les résultats actuels obtenus avec un certain réseau personel, plus d'utilisateurs sont alors considérés.

Nous évaluons DT^2 à partir de traces réelles de *CiteULike* et *delicious*, impliquant 10,000 et 50,000 utilisateurs respectivement. DT^2 surpasse d'autres approches de traitement des requêtes

top-k basées sur des tags, personnalisé ou non. Par exemple, avec des réseaux personnels de 25 utilisateurs de *delicious* dans DT^2, pour les requêtes corrélées aux profils, la qualité des résultats est à peu près semblable à ceux obtenus avec des réseaux personnels des 500 utilisateurs en personnalisation hors ligne, et jusqu'à 47% mieux par rapport à une approche non-personnalisée de plus de 50,000 utilisateurs (figure 3.7). Pour les requêtes non-corrélées, la qualité des résultats est plus de 64% meilleure qu'avec la personnalisation hors ligne, et jusqu'à 46% de plus que sans personnalisation. En outre, le temps nécessaire pour obtenir les k_2 objets avec 25 utilisateurs est toujours plus court que dans une approche non-personnalisée (figure 3.9). Bien que DT^2 prenne un peu plus de temps que une approche personnlisée hors ligne, DT^2 apporte un gain significatif sur la qualité des résultats. DT^2 nécessite environ 99,9% (respectivement 60%) de moins d'espace qu'une approche personnalisée hors ligne (respectivement une approche non-personnalisée). En fait, DT^2 fournit des résultats de haute qualité pour les intérêts ordinaires et émergents, en réduisant le traitement en termes de stockage et de temps.

L'algorithme DT^{2}P^2 Pour appliquer la personnalisation en ligne dans les systèmes pair-à-pair, nous proposons également l'algorithme DT^{2}P^2. En DT^{2}P^2, les utilisateurs s'auto-organisent en un overlay multi-objectif qui permet un traitement des requêtes top-k efficace d'une manière flexible. La partie "stable" de l'overlay a pour but de maintenir les connaissances durables de chaque utilisateur sur le système, c'est à dire l'ensemble des utilisateurs ayant des profils similaires. Ces utilisateurs constituent le réseau d'offre de chaque utilisateur. La partie "souple" de l'overlay est en revanche maintenue lors de la requête et associe à chaque interrogateur, les utilisateurs qui peuvent servir pour le traitement de sa requête en fonction de son intérêt hybride défini par la personnalisation en ligne. Ces utilisateurs constituent le réseau de demande de chaque utilisateur et de sa requête. Les deux parties de l'overlay sont maintenus dynamiquement par un protocole de gossip et le calcul des similarités entre les utilisateurs effectué d'une façon périodique.

Dans DT^{2}P^2, une requête est traitée au sein du réseau de demande de son interrogateur. Quand une requête est émise, l'interrogateur construit progressivement son réseau de demande en communiquant avec les voisins dans son réseau de demande. Chaque utilisateur qui reçoit le message de gossip envoie un sous-ensemble des profils dans son réseau d'offre afin de contribuer au traitement de la requête. L'interrogateur affine son réseau par le calcul des similarités entre son intérêt hybride et les profils des utilisateurs reçus. Les résultats sont ensuite affinés par le traitement de la requête au sein de chaque réseau de demande intermédiaire jusqu'à ce que l'interrogateur soit satisfait ou que le réseau de demande converge.

Afin d'accélérer le traitement des requêtes, en particulier pour les requêtes qui sont peu corrélées au profil de l'interrogateur, un cache de réseaux de demande est maintenu pour chaque utilisateur. Un nombre limité de réseaux de demande, où l'union des requêtes correspondantes couvre le nombre maximum de tags, sont maintenus dans le cache afin de maximiser l'utilité du cache et de minimiser l'impact négatif de la dynamique. Une durée de vie (TTL) est également utilisée pour contrôler l'âge de chaque réseau de demande dans le cache. Au moment de la requête, l'utilisateur choisit parmi son réseau d'offre et ceux dans son cache, le réseau le plus susceptible de répondre à sa requête comme réseau de demande initial et l'affine progressivement.

Nous évaluons DT^{2}P^2 en utilisant les mêmes ensembles de données de *CiteULike* et *delicious* que ceux utilisés pour l'évaluation de DT^2. Notre évaluation montre que le traitement personnalisé des requêtes en ligne peut être réalisé efficacement si l'interrogateur affine progressivement son réseau de demande par gossip. Les mêmes résultats que ceux obtenus dans un système centralisé (DT^2)

avec l'information globale peuvent être obtenus en 15 cycles de gossips (figure 3.14). Si le cache, contenant au plus 5 réseaux de demande en cache, est utilisé, les résultats de top-10 s'améliorent de plus de 20% pour les premiers cycles (figure 3.18). Si la requête correspond aux intérêts émergents de l'interrogateur, cette amélioration peut aller jusqu'à 2,8 fois plus (figure 3.19). Le cache garantit également de meilleurs résultats (10% par cycle) lorsque les intérêts des utilisateurs changent (figure 3.21). Il y a un compromis entre le temps de réponse et la consommation de bande passante parce que la quantité de données transmises lors de chaque cycle est fixée. En supposant que l'utilisateur gossipe chaque seconde pour affiner son réseau de demande, ce qui le donne suffisamment de temps pour évaluer les résultats intermédiaires, 15 secondes serait suffisant pour répondre à une requête. Cela nécessite, en moyenne, une bande passante de 63 kilo-octets par seconde. Plus de bande passante accélérerait le traitement des requêtes probablement. Pour résumer, DT^2P^2 réalise la personnalisation en ligne par gossip d'une manière entièrement décentralisée.

Conclusions et perspectives

Cette thèse propose de nouveaux algorithmes permettant d'effectuer des recherches personnalisées de manière efficace dans des systèmes dynamiques, centralisés ou décentralisés, selon deux axes majeurs : (i) la personnalisation hors ligne qui s'appuie sur le comportement passé des utilisateurs et (ii) la personnalisation en ligne qui s'appuie sur le comportement passé et la requête en cours.

Nous présentons d'abord l'algorithme P3K, qui décentralise une approche existante et réalise le traitement personnalisé des requêtes top-k hors ligne dans les systèmes pair-à-pair. Ensuite, nous présentons P4Q, une extension de P3K qui améliore les performances du système en termes de stockage, bande passante et la robustesse en distribuant le traitement des requêtes. Nos évaluations analytiques et expérimentales démontrent leur efficacité pour le traitement des requêtes top-k, y compris dans les systèmes dynamiques, et montrent en particulier que la capacité inhérente de P4Q à faire face aux mises à jours des profils des utilisateurs.

Dans le but d'améliorer encore la qualité des résultats pour les requêtes représentant les intérêts émergents des utilisateurs, et donc non représentés dans son profil, nous proposons un modèle hybride d'intérêt, prenant en compte à la fois le profil des utilisateurs mais également la requête elle-même. Nous proposons une solution à la fois en centralisée, l'algorithme DT^2, qui effectue une recherche de type top-k à deux reprises. L'algorithme DT^2P^2, exécute efficacement la personnalisation en ligne de manière entièrement décentralisée. Les résultats expérimentaux montrent que la personnalisation en ligne est prometteuse pour répondre aux préférences diverses des utilisateurs.

Pour résumer, la contribution de cette thèse réside dans (i) la réalisation du traitement personnalisé des requêtes hors ligne dans les systèmes pair-à-pair à grande échelle; (ii) la proposition de la personnalisation en ligne ainsi que d'une réalisation centralisée; (iii) la réalisation de la personnalisation en ligne dans les systèmes pair-à-pair à grande échelle. Mais plusieurs pistes sont possibles pour mieux soutenir le traitement des requêtes personnalisé, en particulier dans systèmes pair-à-pair.

Tout d'abord, la surveillance et le stockage de tout type d'activité des utilisateurs est une question sensible. Dans les approches centralisées, le comportement de chaque utilisateur est connu par le serveur central, mais il n'y a pas d'association explicite entre un utilisateur et ses intérêts à la disposition des autres utilisateurs. Une approche décentralisée empêche une autorité centrale de faire un usage commercial des profils de tous les utilisateurs, mais susceptible de révéler les profils des utilisateurs à d'autres utilisateurs. En effet, les algorithmes proposés (P3K, P4Q et DT^2P^2) s'appuient sur les utilisateurs qui échangent des profils d'une manière pair-à-pair afin de maintenir

le réseau personnel et traiter les requêtes. Pour assurer l'anonymat de chaque utilisateur, on peut penser à la transmission du profil, de la requête et des résultats partiels par certains utilisateurs choisis au hasard dans le système. Certaines stratégies de chiffrement doivent être considérées afin de s'assurer que les utilisateurs choisis ne sont pas en mesure d'associer l'identité d'utilisateur avec le profil, la requête ou les résultats partiels. Cela représente une perspective intéressante qui peut être explorée.

En outre, les algorithmes proposés dans cette thèse ont pour objectif de traitement des requêtes instantanées, c'est à dire chaque requête est traitée une seule fois avec les informations disponibles dans le système au moment de la requête. Pourtant, dans les applications de Web 2.0, ainsi que des systèmes collaboratifs, des nouveaux contenus sont créés et partagés à un rythme très élevé. En conséquence, les utilisateurs peuvent être disposés à recevoir les résultats les plus récents ayant trait à leurs requêtes. La recherche persistante permet aux utilisateurs d'émettre la requête une fois et de recevoir des mises à jour en temps réel chaque fois qu'il y a un nouveau résultat pour cette requête. En DT^2P^2, les requêtes personnalisées sont traitées en ligne. Un utilisateur "*bouge*" virtuellement dans le système pair-à-pair en affinant son réseau de demande, ce qui lui permet d'approcher les résultats souhaités progressivement. Il est naturel d'étendre cette idée au traitement des requêtes persistantes. Le réseau de demande peut être maintenu aussi longtemps que l'utilisateur ne supprime pas la requête. Mettre cette idée en pratique est toutefois difficile : plusieurs questions importantes doivent être considérées comme par exemple, en plus du profil et de la requête, de nombreux autres facteurs pourraient être pris en compte pour la sélection des voisins pour le réseau de demande, y compris la qualité des résultats de la requête en cours, la fréquence d'arrivée des nouveaux résultats, le niveau d'activité des voisins, etc. En plus, un utilisateur peut émettre plusieurs requêtes persistantes au cours d'une même période. Il serait intéressant d'enquêter sur certains mécanismes qui exploitent la similarité entre les requêtes et d'affiner leurs réseaux de demande d'une manière plus efficace afin de réduire les dépenses non nécessaires en termes de stockage et de bande passante.

LIST OF FIGURES

LIST OF TABLES

BIBLIOGRAPHY

[1] Facebook. Facebook Statistics. http://www.facebook.com/press/info.php?statistics, 2010.

[2] Twitter Blog. Big Goals, Big Game, Big Records. http://blog.twitter.com/2010/06/big-goals-big-game-big-records.html, June 2010.

[3] YouTube. 24 hours of video uploaded to YouTube every minute. http://goo.gl/PxfU, March 2010.

[4] Lev Grossman. Person of the Year: You. http://www.time.com/time/covers/0,16641,20061225,00.html, December 2006.

[5] Claudine Beaumont. New York plane crash: Twitter breaks the news, again. http://www.telegraph.co.uk/technology/twitter/4269765/New-York-plane-crash-Twitter-breaks-the-news-again.html, Jan 2009.

[6] Matt McGee. By the numbers: Twitter vs. facebook vs. google buzz. http://goo.gl/Kwxl, Feb 2010.

[7] BBC. Twitter tweets are 40% 'babble'. http://news.bbc.co.uk/2/hi/technology/8204842.stm, August 2009.

[8] Chris Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, New York, 2006.

[9] PPS. http://www.pps.tv, 2010.

[10] The Inquisitr. Skype Commands 13 Percent of International Phone Calls. http://goo.gl/vrUT, May 2010.

[11] John Robb. Web 2.0. http://jrobb.mindplex.org/2003/08/16.html, August 2003.

[12] Sihem Amer-Yahia, Michael Benedikt, Laks V. S. Lakshmanan, and Julia Stoyanovich. Efficient network aware search in collaborative tagging sites. *Proceedings of the VLDB Endowment*, 1(1):710–721, 2008.

[13] delicious. http://www.delicious.com.

[14] Paul Heymann, Georgia Koutrika, and Hector Garcia-Molina. Can social bookmarking improve web search? In *Proceedings of the International Conference on Web Search and Web Data Mining (WSDM '08)*, pages 195–206, 2008.

[15] Delicious blog. Delicious is 5! `http://blog.delicious.com/blog/2008/11/delicious-is-5.html`, November 2008.

[16] Citeulike. `http://www.citeulike.org/`.

[17] Flickr. `http://www.flickr.com`.

[18] Peter Forret. A picture a day: Flickr's storage growth. `http://blog.forret.com/2006/10/a-picture-a-day-flickrs-storage-growth/`, October 2006.

[19] Heather Champ. 4,000,000,000. `http://blog.flickr.net/en/2009/10/12/4000000000/`, October 2009.

[20] Ed H. Chi and Todd Mytkowicz. Understanding the efficiency of social tagging systems using information theory. In *Proceedings of the nineteenth ACM Conference on Hypertext and Hypermedia (HT '08)*, pages 81–88, 2008.

[21] Harry Halpin, Valentin Robu, and Hana Shepherd. The complex dynamics of collaborative tagging. In *Proceedings of the sixteenth International Conference on World Wide Web (WWW '07)*, pages 211–220, 2007.

[22] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing (PODC '87)*, pages 1–12, 1987.

[23] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[24] Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed systems*, 14(3):248–258, 2003.

[25] Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulieacute. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37:60–67, 2004.

[26] Márk Jelasity, Voulgaris Spyros, Guerraoui Rachid, Kermarrec Anne-Marie, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

[27] Spyros Voularis, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Network and Systems Management*, 13(2):197–217, 2005.

[28] Spyros Voulgaris and Maarten Van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Proceedings of the eleventh International Euro-Par Conference (EuroPar'05)*, pages 1143–1152. Springer, 2005.

[29] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the fourth-fourth annual IEEE Symposium on Foundations of Computer Science (FOCS '03)*, page 482, Washington, DC, USA, 2003. IEEE Computer Society.

[30] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252, 2005.

[31] Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Proceedings of the twenty-fifth annual ACM Symposium on Principles of Distributed Computing (PODC '06)*, pages 123–132, 2006.

[32] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection. In *Proceedings of the International Conference on Middleware (Middleware'98)*, pages 55–70, 1998.

[33] Sridharan Ranganathan, Alan D. George, Robert W. Todd, and Matthew C. Chidester. Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing*, 4(3):197–209, 2001.

[34] Katherine Guo, Mark Hayden, Robbert van Renesse, Werner Vogels, and Kenneth P. Birman. Gsgc: An efficient gossip-style garbage collection scheme for scalable reliable multicast. Technical report, Cornell University, Ithaca, NY, USA, 1997.

[35] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. *Lecture Notes in Computer Science (LNCS)*, 2977:265–282, 2004.

[36] Alberto Montresor, Mark Jelasity, and Ozalp Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 19, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In *Proceedings of the second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Lecture Notes on Computer Science, pages 256–267. Springer-Verlag, 2003.

[38] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia systems*, 9(2):170–184, 2003.

[39] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. *IEEE/ACM Transactions on Networking*, 12(2):219–232, 2004.

[40] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the fifth ACM/IFIP/USENIX International Conference on Middleware (Middleware '04)*, pages 79–98, 2004.

[41] Paul. Erdos and Alfred. Renyi. On the evolution of random graphs. *Publications of the Mathmatical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[42] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 53(13):2321–2339, 2009.

[43] Spyros Voulgaris, Maarten Van Steen, and Konrad Iwanicki. Proactive gossip-based management of semantic overlay networks: Research articles. *Concurrency and Computation: Practice & Experience*, 19(17):2299–2311, 2007.

[44] Patrick T. Eugster, Rachid Guerraoui, Sidath B. Handurukande, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, 2003.

[45] Rena Bakhshi, Daniela Gavidia, Wan Fokkink, and Maarten van Steen. An analytical model of information dissemination for a gossip-based protocol. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 53(13):2288–2303, 2009.

[46] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '01)*, pages 102–113, 2001.

[47] Ronald Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record*, 31(2):109–118, 2002.

[48] Surya Nepal and M. V. Ramakrishna. Query processing issues in image(multimedia) databases. In *Proceedings of the fifteenth International Conference on Data Engineering (ICDE'99)*, page 22, Washington, DC, USA, 1999. IEEE Computer Society.

[49] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kiessling. Optimizing multi-feature queries for image databases. In *VLDB'00: Proceedings of the twenty-sixth International Conference on Very Large Data Bases*, pages 419–428, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[50] Ronald Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58(1):83–99, 1999.

[51] Edward L. Wimmers, Laura M. Haas, Mary T. Roth, and Christoph Braendli. Using fagin's algorithm for merging ranked results in multimedia middleware. In *Proceedings of the fourth IECIS International Conference on Cooperative Information Systems (COOPIS '99)*, page 267, Washington, DC, USA, 1999. IEEE Computer Society.

[52] Ulrich Gü ntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC '01)*, pages 622 –628, April 2001.

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

[54] Jaime Teevan, Susan T. Dumais, and Eric Horvitz. Characterizing the value of personalizing search. In *Proceedings of the thirtieth annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '07)*, pages 757–758, 2007.

[55] Susan Gauch, Jason Chaffee, and Alaxander Pretschner. Ontology-based personalized search and browsing. *Web Intelligence and Agent SystemsWeb Intelligence and Agent Systems*, 1(3-4):219–234, 2003.

[56] Paul Alexandru Chirita, Wolfgang Nejdl, Raluca Paiu, and Christian Kohlschütter. Using odp metadata to personalize search. In *Proceedings of the twenty-eighth annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '05)*, pages 178–185, 2005.

[57] Open directory project. `http://www.dmoz.org/`.

[58] John M. Carroll and Mary Beth Rosson. Paradox of the active user. *Interfacing thought: cognitive aspects of human-computer interaction*, pages 80–111, 1987.

[59] Fang Liu, Clement Yu, and Weiyi Meng. Personalized web search by mapping user queries to categories. In *Proceedings of the eleventh International Conference on Information and Knowledge Management (CIKM '02)*, pages 558–565, 2002.

[60] Kazunari Sugiyama, Kenji Hatano, and Masatoshi Yoshikawa. Adaptive web search based on user profile constructed without any effort from users. In *Proceedings of the thirteenth International Conference on World Wide Web (WWW '04)*, pages 675–684, 2004.

[61] Xuehua Shen, Bin Tan, and ChengXiang Zhai. Implicit user modeling for personalized search. In *Proceedings of the fourteenth ACM International Conference on Information and Knowledge Management (CIKM '05)*, pages 824–831, 2005.

[62] Feng Qiu and Junghoo Cho. Automatic identification of user interest for personalized search. In *Proceedings of the fifteenth international conference on World Wide Web (WWW '06)*, pages 727–736, 2006.

[63] Michael G. Noll and Christoph Meinel. Web search personalization via social bookmarking and tagging. In *Proceedings of the sixth International Semantic Web and second Asian Conference on Asian Semantic Web Conference (ISWC'07/ASWC'07)*, pages 367–380, Berlin, Heidelberg, 2007. Springer-Verlag.

[64] Micro Speretta and Susan Gauch. Personalized search based on user search histories. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI '05)*, pages 622–628, Washington, DC, USA, 2005. IEEE Computer Society.

[65] Alan Mislove, Krishna P. Gummadi, and Peter Druschel. Exploiting social networks for internet search. In *Proceedings of the fifth Workshop on Hot Topics in Networks (HotNets'06)*, 2006.

[66] Ralf Schenkel, Tom Crecelius, Mouna Kacimi, Sebastian Michel, Thomas Neumann, Josiane X. Parreira, and Gerhard Weikum. Efficient top-k querying over social-tagging networks. In *Proceedings of the thirty-first annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '08)*, pages 523–530, 2008.

[67] Julia Stoyanovich, Sihem Amer-Yahia, Cameron Marlow, and Cong Yu. A study of the benefit of leveraging tagging behavior to model users' interests in del.icio.us. In *In AAAI Spring Symposium on Social Information Processing*, 2008.

[68] Aleksandra K. Milicevic, Alexandros Nanopoulos, and Mirjana Ivanovic. Social tagging in recommender systems: a survey of the state-of-the-art and possible extensions. *Artificial Intelligence Review*, 33(3):187–209, 2010.

[69] Jaime Teevan, Meredith R. Morris, and Steve Bush. Discovering and using groups to improve personalized search. In *Proceedings of the second ACM International Conference on Web Search and Data Mining (WSDM '09)*, pages 15–24, 2009.

[70] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[71] Will Hill, Larry Stead, Mark Rosenstein, and George Furnas. Recommending and evaluating choices in a virtual community of use. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*, pages 194–201, 1995.

[72] Zhicheng Dou, Ruihua Song, and Ji-Rong Wen. A large-scale evaluation and analysis of personalized search strategies. In *Proceedings of the sixteenth International Conference on World Wide Web (WWW '07)*, pages 581–590, 2007.

[73] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the twenty-third annual ACM Symposium on Principles of Distributed Computing (PODC '04)*, pages 206–215, 2004.

[74] Hailing Yu, Hua-Gang Li, Ping Wu, Divyakant Agrawal, and Amr El Abbadi. Efficient processing of distributed top-k queries. In Kim Viborg Andersen, John Debenham, and Roland Wagner, editors, *Database and Expert Systems Applications*, volume 3588 of *Lecture Notes in Computer Science*, pages 65–74. Springer Berlin, 2005.

[75] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Klee: a framework for distributed top-k query algorithms. In *Proceedings of the thirty-first International Conference on Very Large Data Bases (VLDB '05)*, pages 637–648. VLDB Endowment, 2005.

[76] Thomas Neumann, Matthias Bender, Sebastian Michel, Ralf Schenkel, Peter Triantafillou, and Gerhard Weikum. Optimizing distributed top-k queries. In *Proceedings of the ninth International Conference on Web Information Systems Engineering (WISE '08)*, pages 337–349, Berlin, Heidelberg, 2008. Springer-Verlag.

[77] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, pages 161–172, 2001.

[78] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, pages 149–160, 2001.

[79] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (Middleware '01)*, pages 329–350, London, UK, 2001. Springer-Verlag.

[80] Chunqiang Tang, Zhichen Xu, and Mallik Mahalingam. psearch: information retrieval in structured overlays. *SIGCOMM Computer Communication Review*, 33(1):89–94, 2003.

[81] Chunqiang Tang and Sandhya Dwarkadas. Hybrid global-local indexing for effcient peer-to-peer information retrieval. In *Proceedings of the first Conference on Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[82] Matthias Bender, Sebastian Michel, Peter Triantafillou, Gerhard Weikum, and Christian Zimmer. Minerva: collaborative p2p search. In *Proceedings of the thirty-first International Conference on Very Large Data Bases (VLDB '05)*, pages 1263–1266. VLDB Endowment, 2005.

[83] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Processing top-k queries in distributed hash tables. In *Proceedings of the thirteenth European International Conference on Parallel Processing (Euro-Par '07)*, pages 489–502, Berlin, Heidelberg, 2007. Springer-Verlag.

[84] Michael W. Berry, Zlatko Drmac, and Elizabeth R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[85] Gnutella specification. `http://wiki.limewire.org/index.php?title=GDF`.

[86] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Reducing network traffic in unstructured p2p systems using top-k queries. *Distributed and Parallel Databases*, 19(2-3):67–86, 2006.

[87] Francisco Matias Cuenca-Acuna, Christopher Peery, Richard P. Martin, and Thu D. Nguyen. Planetp: Using gossiping to build content addressable peer-to-peer information sharing communities. In *Proceedings of the twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC '03)*, page 236, Washington, DC, USA, 2003. IEEE Computer Society.

[88] Vicent Cholvi, Pascal Felber, and Ernst Biersack. Efficient search in unstructured peer-to-peer networks. In *Proceedings of the sixteenth annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04)*, pages 271–272, 2004.

[89] Andrew Fast, David Jensen, and Brian Neil Levine. Creating social networks to improve peer-to-peer networking. In *Proceedings of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05)*, pages 568–573, 2005.

[90] Yann Busnel and Anne-Marie Kermarrec. Proxsem: Interest-based proximity measure to improve search efficiency in p2p systems. In *Proceedings of the fourth European Conference on Universal Multiservice Networks (ECUMN'2007)*, Toulouse, France, February 2007.

[91] Hai Jin, Xiaomin Ning, and Hanhua Chen. Efficient search for peer-to-peer information retrieval using semantic small world. In *Proceedings of the fifteenth International Conference on World Wide Web (WWW '06)*, pages 1003–1004, 2006.

[92] Johan A. Pouwelse, Pawel Garbacki, Jun Wang, Arno Bakker, J. Yang, Alexandru Iosup, Dick H. J. Epema, Marcel J. T. Reinders, Maarten R. Van Steen, and Henk J. Sips. Tribler: a social-based peer-to-peer system. *Concurrency and Computation: Practice & Experience*, 20(2):127–138, 2008.

[93] Gang Chen, Chor Ping Low, and Zhonghua Yang. Enhancing search performance in unstructured p2p networks based on users' common interest. *IEEE Transactions on Parallel and Distributed Systems*, 19(6):821–836, 2008.

[94] Wolf-Tilo Balke, Wolfgang Nejdl, Wolf Siberski, and Uwe Thaden. Progressive distributed top-k retrieval in peer-to-peer networks. In *Proceedings of the twenty-first International Conference on Data Engineering (ICDE '05)*, pages 174–185, Washington, DC, USA, 2005. IEEE Computer Society.

[95] Keping Zhao, Yufei Tao, and Shuigeng Zhou. Efficient top-k processing in large-scaled distributed environments. *Data and Knowledge Engineering*, 63(2):315–335, 2007.

[96] Akrivi Vlachou, Christos Doulkeridis, Kjetil Nørvåg, and Michalis Vazirgiannis. On efficient top-k query processing in highly distributed environments. In *Proceedings of the 2008 ACM SIGMOD International Monference on Management of data (SIGMOD '08)*, pages 753–764, 2008.

[97] Paul Alexandru Chirita, Paul Alex, Ru Chirita, Wolfgang Nejdl, and Oana Scurtu. Knowing where to search: Personalized search strategies for peers in p2p networks. In *Proceedings of the P2P Information Retrieval Workshop at the twenty-seventh International ACM SIGIR Conference*, 2004.

[98] Frédéric Dang Ngoc, Joaquín Keller, and Gwendal Simon. Maay: a decentralized personalized search system. In *International Symposium on Applications and the Internet*, pages 8 pp. – 179, 2006.

[99] Matthias Bender, Tom Crecelius, Mouna Kacimi, Sebastian Michel, Josiane Xavier Parreira, and Gerhard Weikum. Peer-to-peer information search: Semantic, social, or spiritual? *IEEE Data Engineering Bulletin*, 30(2):51–60, 2007.

[100] Alimohammad Saghiri and Alireza Bagheri. An adaptive architecture for personalized search ?engine in ubiquitous environment with peer to peer systems. In *International Conference on Information and Multimedia Technology (ICIMT '09)*, pages 107 –111, 2009.

[101] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the twelfth International Conference on World Wide Web (WWW '03)*, pages 271–279, 2003.

[102] Solomon Kullback. *Information Theory and Statistics*. Wiley, New York, 1959.

[103] Shengliang Xu, Shenghua Bao, Ben Fei, Zhong Su, and Yong Yu. Exploting folksonomy for personalized search. In *Proceedings of the thirty-first annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08)*, pages 155–162, 2008.

[104] Xin Li, Lei Guo, and Yihong Eric Zhao. Tag-based social interest discovery. In *Proceeding of the seventeenth International Conference on World Wide Web (WWW '08)*, pages 675–684, 2008.

[105] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[106] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the seventh ACM SIGCOMM Conference on Internet Measurement (IMC '07)*, pages 29–42, 2007.

[107] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, CA, 2 edition, 1999.

[108] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM*, 13(7):422–426, 1970.

[109] Gerard Salton, Andrew Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[110] Robert M. Bell and Yehuda Koren. Improved neighborhood-based collaborative filtering. In *Proceedings of the KDD-Cup Workshop at the thirteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2007.

[111] Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice*. Addison Wesley, 1 edition, February 2009.

[112] Jonathan Gemmell, Thomas Schimoler, Maryam Ramezani, and Bamshad Mobasher. Adapting k-nearest neighbor for tag recommendation in folksonomies. In *Proceedings of seventh Workshop on Intelligent Techniques for Web Personalization and Recommender Systems*, volume 528 of *CEUR Workshop Proceedings*, 2009.

[113] Benjamin Markines, Ciro Cattuto, Filippo Menczer, Dominik Benz, Andreas Hotho, and Gerd Stumme. Evaluating similarity measures for emergent semantics of social tagging. In *Proceedings of the eighteenth International conference on World Wide Web (WWW '09)*, pages 641–650, 2009.

[114] Jian-Tao Sun, Hua-Jun Zeng, Huan Liu, Yuchang Lu, and Zheng Chen. Cubesvd: a novel approach to personalized web search. In *Proceedings of the fourteenth International Conference on World Wide Web (WWW '05)*, pages 382–390, 2005.

[115] Tie-Yan Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[116] Stephen E. Robertson and Stephen G. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the seventeenth annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '94)*, pages 232–241, 1994.

[117] Xiao Bai, Marin Bertier, Rachid Guerraoui, Anne-Marie Kermarrec, and Vincent Leroy. Gossiping personalized queries. In *Proceedings of the thirteenth International Conference on Extending Database Technology (EDBT '10)*, pages 87–98, 2010.

[118] Andreas Loupasakis and Peter Triantafillou. eXo: Decentralized social networking with autonomy, scalability, and efficiency. In *Proceedings of the Hellenic Data Management Symposium (HDMS'10)*, 2010.

[119] Xiao Bai, Marin Bertier, Rachid Guerraoui, and Anne-Marie Kermarrec. Toward personalized peer-to-peer top-k processing. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems(SNS '09)*, pages 1–6, 2009.

[120] Neal E. Young. On-line file caching. In *Proceedings of the ninth annual ACM-SIAM Symposium on Discrete Algorithms (SODA '98)*, pages 82–86, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[121] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems (USITS'97)*, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[122] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the fourteenth International Conference on World Wide Web (WWW '05)*, pages 257–266, 2005.

[123] Yohannes Tsegay, Andrew Turpin, and Justin Zobel. Dynamic index pruning for effective caching. In *Proceedings of the sixteenth ACM Conference on Information and Knowledge Management (CIKM '07)*, pages 987–990, 2007.

[124] Ricardo Baeza-Yates, Aristides Gionis, Flavio P. Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. Design trade-offs for search engine caching. *ACM Transactions on the Web (TWEB)*, 2(4):1–28, 2008.

[125] Marin Bertier, Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Vincent Leroy. The gossple anonymous social network. In *Proceedings of the eleventh International Middleware Conference (Middleware '10)*, pages 1–23, 2010.

[126] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

Résumé

La révolution Web 2.0 a transformé l'Internet, une infrastructure auparavant en lecture seule, en une plate-forme collaborative en lecture-écriture. La forte augmentation des donnés générées par les utilisateurs des systèmes collaboratifs constitue désormais une source considérable d'informations. Pourtant, effectuer efficacement des recherches dans un tel environnement est devenu plus difficile, en particulier lorsque ces recherches engendrent des ambiguïtés. Personnaliser les recherches permet d'éviter ces écueils en limitant les recherches au sein d'un réseau très réduit de participants ayant des intérêts similaires. Toutefois, les solutions centralisées pour mettre en œuvre cette personnalisation s'avèrent difficile compte tenu du volume important d'informations qui doit être maintenu pour chaque utilisateur. La nature dynamique de ces systèmes, dans lesquels les utilisateurs changent potentiellement souvent d'intérêt, complique la tâche.

Cette thèse propose de nouveaux algorithmes permettant d'effectuer des recherches personnalisées de manière efficace dans des systèmes dynamiques, centralisés ou décentralisés, selon deux axes majeurs : (i) la personnalisation hors ligne qui s'appuie sur le comportement passé des utilisateurs et (ii) la personnalisation en ligne qui s'appuie sur le comportement passé et la requête en cours.

Nous présentons d'abord l'algorithme P3K, qui décentralise une approche existante et réalise le traitement personnalisé des requêtes top-k hors ligne dans les systèmes pair-à-pair. Ensuite, nous présentons P4Q, une extension de P3K qui améliore les performances du système en termes de stockage, bande passante et la robustesse en distribuant le traitement des requêtes. Les deux algorithmes, P3K et P4Q, reposent sur des protocoles épidémiques pour capturer la similarité implicite entre les utilisateurs et associer ainsi à chaque utilisateur un "réseau personnel" dans lequel traiter la requête. Nos évaluations analytiques et expérimentales démontrent leur efficacité pour le traitement des requêtes top-k, y compris dans les systèmes dynamiques, en particulier que la capacité inhérente de P4Q à faire face aux mises à jours des profils des utilisateurs.

Dans le but d'améliorer encore la qualité des résultats pour les requêtes représentant les intérêts émergents des utilisateurs, et donc non représentés dans son profil, nous proposons un modèle hybride d'intérêt, prenant en compte à la fois le profil des utilisateurs mais également la requête elle-même. Nous avons proposé une solution à la fois en centralisé, l'algorithme DT^2, qui effectue une recherche de type top-k à deux reprises. L'algorithme DT^2P^2, exécute efficacement la personnalisation en ligne de manière entièrement décentralisée. Les résultats expérimentaux sur des traces réelles de systèmes collaboratifs, montrent que la personnalisation en ligne est prometteuse pour répondre aux préférences diverses des utilisateurs.

Abstract

The Web 2.0 revolution has transformed the Internet from a read-only infrastructure to an active read-write platform. The rapid increasing amount user-generated content in collaborative tagging systems provides a huge source of information. Yet, performing effective search becomes more challenging, especially when we seek the most appropriate items that match a potentially ambiguous query. Personalization is appealing in this context as it limits the search for the items within a small network of participants with similar interests. However, centralized solutions for this personalization do not scale 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 change their profiles by tagging new items.

In this regard, this thesis deals with the efficiency and scalability of personalized query processing, from centralized to decentralized systems, around two axes: (i) the off-line personalization that relies on users' past tagging behaviors and (ii) the on-line personalization that relies on both the past behaviors and the current query.

We first present the algorithm P3K, which decentralizes a state-of-the-art approach and achieves off-line personalized top-k processing in peer-to-peer systems. Then we present P4Q, an extension of P3K that enhances the system performance in terms of storage, bandwidth and robustness. Both P3K and P4Q rely on gossip-based protocols to capture the implicit similarity between users and associate each user with a set of social acquaintances to process the query. Analytical and experimental evaluations convey their scalability and efficiency for top-k query processing, as well as the inherent ability of P4Q to cope with users updating profiles and departing.

To further improve the result quality for the queries depicting emerging interests of the querier, we propose a hybrid interest model, taking into account both the tagging profile and the query, to perform personalized query processing. This is achieved on-line in a centralized system by doing top-k twice with the algorithm DT^2. Then we propose the algorithm DT^2P^2 that efficiently performs the same on-line personalization with improved scalability in a fully decentralized system. Experimental results on real datasets show that on-line personalization is promising to fulfill the diverse user preferences while the proposed algorithms make it feasible in both centralized and decentralized systems.

N°d'ordre : XXXXXXXX

Institut National des Sciences Appliquées de Rennes

20, Avenue des Buttes de Coësmes . CS 14315 . F-35043 Rennes Cedex

Tel : 02 23 23 82 00 – Fax : 02 23 23 83 96