Efficient Network-Aware Search in Collaborative Tagging Sites

Sihem Amer Yahia
Yahoo! Research
sihem@yahoo-inc.com

Michael Benedikt
Oxford University
benedikt@comlab.ox.ac.uk

Laks V.S. Lakshmanan*
University of British Columbia
laks@cs.ubc.ca

Julia Stoyanovich†
Columbia University
jds1@cs.columbia.edu

ABSTRACT

The popularity of collaborative tagging sites presents a unique opportunity to explore keyword search in a context where query results are determined by the opinion of a network of taggers related to a seeker. In this paper, we present the first in-depth study of *network-aware* search. We investigate efficient top-k processing when the score of an answer is computed as its popularity among members of a seeker's network. We argue that obvious adaptations of top-k algorithms are too space-intensive, due to *the dependence of scores on the seeker's network*. We therefore develop algorithms based on maintaining *score upper-bounds*. The *global upper-bound approach* maintains a single score upper-bound for every pair of item and tag, over the entire collection of users. The resulting bounds are very coarse. We thus investigate *clustering seekers based on similar behavior of their networks*. We show that finding the optimal clustering of seekers is intractable, but we provide heuristic methods that give substantial time improvements. We then give an optimization that can benefit smaller populations of seekers based on *clustering of taggers*. Our results are supported by extensive experiments on del.icio.us datasets.

1. INTRODUCTION

The unprecedented popularity of collaborative tagging sites such as CiteULike for academic papers, del.icio.us for webpages, Flickr and Snapfish for photos and YouTube for videos, presents a unique opportunity for incorporating *social behavior* into processing search queries. In del.icio.us, users bookmark and tag URLs, form social ties with others, and subscribe to their friends' feeds to discover which URLs are being bookmarked. Although browsing by tag or by network is currently the predominant way of reaching content on these sites, ranked keyword-based search is supported as well, and search will become more important as the size of networks and tagged content expand. An important question then is how to support *network-aware search* – a mechanism that will return the top-ranked answers to a query consisting of a set of tags, given a user with a particular network. Such a mechanism is important not only for supporting search within these sites, but also for incorporating a seeker's network and tagging behavior in general web search. In this paper, we revisit top-k processing in this context.

Information Retrieval (e.g., web search) generally assumes content to be relatively static, while user interests are dynamic, expressed via keyword queries [1]. On the other hand, Publish/Subscribe assumes static publisher needs and dynamic streaming content [20]. In collaborative tagging sites, *both content and interest are dynamic*. Users tag new photos on Flickr and new videos on YouTube every day and develop interest in new topics. We model collaborative tagging sites as follows: users in the system can be either *taggers* or *seekers*. *Taggers* annotate *items* with one or more *tags*; a query is composed of a set of tags and is asked by a *seeker*; a linking relation connects seekers and taggers, thereby forming the *network* associated with each seeker. In practice seekers and taggers may be the same and our model and algorithms will not preclude this possibility.

Given a seeker, a network of taggers, and a query in the form of a set of tags, we wish to return the most relevant items. Relevance of an item should certainly be a function of the number of taggers within the seeker's network who tagged the item with a tag in the query. We formalize network-aware search in collaborative tagging sites and define the problem of efficiently processing top-k queries in the presence of dynamic scores (Section 2). We introduce a general class of scoring functions that reflects this intuition. This class captures the core of the ranking functions desired in many application scenarios, such as hotlists, popularity among friends, and popularity among a network of peers in the same age group or geographic location. Our core question is then *how to achieve efficient top-k processing of these network-aware search queries*.

*Research supported by grant from the Natural Sciences and Engineering Research Council of Canada.

†This research was supported in part by National Institute of Health grant 5 U54 CA121852-03.

VLDB '08, August 24-30, 2008, Auckland, New Zealand

We consider what sort of storage structures are needed to support such queries. In the network-unaware setting, one would use *per-tag inverted lists*, listing items ranked by their score. A naive extension of this is to generate similar lists for each *(tag,seeker)* pair since items have a different score per seeker's network. Items would then be sorted according to their network-aware score in each inverted list. The well-known top-k algorithms [4] could then be directly applied to aggregate over tags in a query. We refer to this storage strategy as `Exact`, since it requires storing exact scores for each item for each *(tag,seeker)* pair. This strategy can clearly benefit from the efficiency of traditional top-k algorithms. However, materializing each of these lists would be prohibitive in terms of space, since there are potentially as many lists per tag as there are seekers. Consequently, we explore *dynamic computation* of scores, given a seeker's network and a tag, as a way of achieving *a balance between processing time and storage space*. Note that while traditional top-k algorithms aggregate scores on different keywords at query time, they assume the inverted lists are ordered by exact scores, even if exact scores of list entries are not available. The most straightforward dynamic approach materializes only per-tag inverted lists. These inverted lists are seeker-independent, so their entries cannot contain exact per-seeker scores but only upper-bounds for each item, i.e., its max score over all seekers. We refer to this strategy as `Global Upper-Bound`. We develop generalizations of top-k processing algorithms to incorporate network-aware search. In particular, we modify the classic `NRA` (No Random Access) and `TA` (Threshold Algorithm) to account for score upper-bounds and dynamic computation of exact scores (Section 3).

The `Exact` and `Global Upper-Bound` strategies represent two extremes in the space-time tradeoff. While `Global Upper-Bound` will offer considerable savings in space, we expect query processing to take much longer, since the upper-bounds may be very coarse. The finer the upper-bounds, the closer they are to exact scores for a given seeker, which affects the order of entries in the inverted lists. We explore two refinements in Section 4.

Our first refinement identifies groups of seekers whose "network-aware" scores are close. The score upper-bound computed for such a group will then be tighter than the global upper-bound. These groups represent seekers who exhibit *similar networking behavior*. We refer to this strategy as `Cluster-Seekers`. This strategy leads to one inverted list per *(tag,cluster)* pair. At query time, we find the *(tag,cluster)* inverted lists associated to that seeker and aggregate over them. We will see that the `Cluster-Seekers` strategy performs faster than `Global Upper-Bound` and consumes less space than `Exact`.

The performance of `Cluster-Seekers` depends on finding good clusterings. We explore quality metrics that measure how well the order of entries in a per-cluster inverted list reflects their order in a seeker-specific inverted list, for seekers who belong to that cluster.

Next, we explore `Cluster-Taggers`, a strategy which aims to group taggers in networks based on *similarity in their tagging behavior*. Each tagger cluster is associated with an inverted list, where items are ranked by their upper-bound within the cluster of taggers. At query time, we determine the tagger clusters associated with a given seeker and compute scores by aggregating over the inverted lists associated with these tagger clusters. Taggers in a seeker's network may belong to different clusters, and so multiple inverted lists per tag may be relevant for a given seeker. We thus may have to aggregate over a larger number of inverted lists, albeit with tighter upper-bounds per list. In our experiments, we found that `Cluster-Taggers` imposes little space penalty compared to `Global Upper-Bound`, and outperforms `Cluster-Seekers` in both space overhead and query time. However, not every seeker will benefit from this approach, e.g., seekers whose taggers fall into many clusters.

We run extensive experiments on datasets from del.icio.us (Section 5). We measure the performance of clustering in terms of space consumption and the performance of top-k algorithms in terms of the number of sequential and random accesses. To the best of our knowledge, *this is the first attempt to incorporate social behavior into indexing and top-k query processing.*

2. DATA MODEL

Our model represents collaborative tagging sites where users have ties with other users and can express their endorsement of visited items – e.g., photos in Flickr and videos in YouTube – by tagging. We model the underlying social network as a directed graph $G = (\mathcal{V}, \mathcal{E})$ whose nodes are users and (directed) edges correspond to ties between them. The edges may correspond to explicit ties such as friendship, similar age group, or people living in the same neighborhood. Alternatively, they may correspond to implicit ties, derived through some further analysis. We describe examples below.

We represent the data relevant to this application using the following relations: (i) $\texttt{Link}(u, v)$ which says there is a directed edge from user u to user v. We assume every user participates in at least one link. (ii) $\texttt{Tagged}(v, i, t)$ which says tagger v tags item i with tag t.

Often we will be interested in iterating over all users in one of the projections of `Link` or `Tagged`. The projection on the first component of `Link` is the `Seekers` set; we will be interested in queries from these users. The projection on the first component of `Tagged` represents the `Taggers` set. We want the tags assigned by these users to influence the way answers to queries are scored. It is convenient to use the following notation. For a seeker $u \in \texttt{Seekers}$, $\texttt{Network}(u)$ is the set of neighbors of u, i.e., $\texttt{Network}(u) = \{v \mid \texttt{Link}(u, v)\}$. We will use the notation $\texttt{Items}(v, t) = \{i \mid \texttt{Tagged}(v, i, t)\}$ to denote the set of items tagged with t by tagger $v \in \texttt{Taggers}$.

2.1 Example Networks

We elaborate on a few natural implicit ties that may exist between users in G. Let $\texttt{Items}(v)$ be the set of items tagged by user v with any tag. Then we could define $\texttt{Link}(u, v)$ to mean that of the items tagged by u, a large fraction are also tagged by v, i.e., $|\texttt{Items}(u) \cap \texttt{Items}(v)|/|\texttt{Items}(u)| > \mathit{thres}$, where thres is a given threshold.

Alternatively, we could define $\texttt{Link}(u, v)$ iff v tags a sufficient fraction of the items tagged by u with the same tag as u, i.e., $|\{i \mid \exists t : \texttt{Tagged}(u, i, t) \wedge \texttt{Tagged}(v, i, t)\}|/|\{i \mid \exists t : \texttt{Tagged}(u, i, t)\}| > \mathit{thres}$.

Figure 1 shows our running example. The network of seeker *Jane* overlaps with *Leia*'s and *Amanda*'s. This is reflected in `Tagged`, where taggers common to *Jane* and *Leia* tag various items.

The techniques of the paper do not assume any particular semantics of how the networks are obtained.

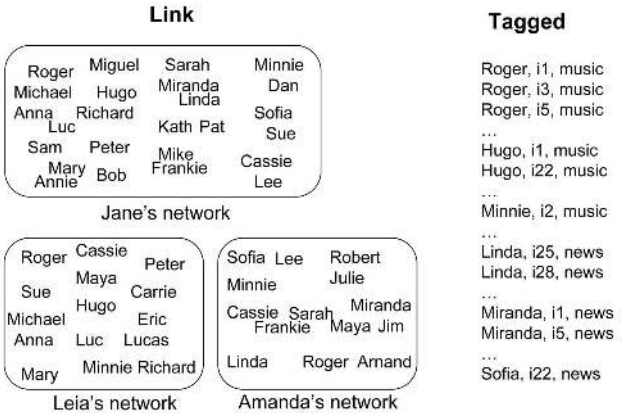


Figure 1: Seekers, Networks and Tagging Actions

2.2 Queries and Scoring

Seekers issue queries in the form of a set of keywords. In order to focus on the aggregation problems rather than the (orthogonal) text matching issues, we treat keywords and tags alike, and our scoring method is based on exact string match. More specifically, given a seeker u and a query $Q = t_1, ..., t_n$, we define the score of an item i for u w.r.t. a tag t_j as a monotone function of the number of taggers in u's network who tagged i with tag t_j, i.e.,
$\texttt{score}_{t_j}(i, u) = f(|\texttt{Network}(u) \cap \{v \mid \texttt{Tagged}(v, i, t_j)\}|)$, where f is a monotone function.

We define the overall query score of an item i for a seeker $u \in \texttt{Seekers}$ as a monotone aggregation of the scores for the individual keywords in the query, i.e.,
$\texttt{score}(i, u) = g(\texttt{score}_{t_1}(i, u), ..., \texttt{score}_{t_n}(i, u))$, where g is a monotone aggregate function.

While the framework is general enough to permit arbitrary monotone functions f and g, we will use $f = count$ and $g = sum$ for ease of exposition. Prior work [18] has shown that when these scoring functions are applied to networks based on implicit common-interest, the resulting lists are good predictors of relevance to the seeker. Hence we use common-interest networks in our experiments.

2.3 Problem Statement

Given a query $Q = t_1, ..., t_n$, issued by user u, and a number k, we want to efficiently determine the top k items, i.e., the k items with the highest overall score.

In the next section, we address the following questions: given the input data modeled using the logical relations `Link` and `Tagged`, what information should we pre-compute in order that well-known top-k algorithms can be leveraged, and how should we adapt these algorithms to work correctly and efficiently in our setting?

3. INVERTED LISTS AND TOP-K

We wish to compute the top-k items which have been tagged by people in a seeker's network with query relevant tags. We organize items in *inverted lists* and study the applicability of typical top-k processing algorithms to our setup.

3.1 Computing Exact Scores

Typically, in Information Retrieval, one inverted list is created for each keyword [1]. Each entry in the list contains the identifier of a document along with its score for that keyword. Storing scores allows to sort entries in the inverted list thereby enabling top-k pruning [6]. While in classic IR each document has a unique score for a keyword (typically, tf*idf [1] or probabilistic [7]), one characteristic of our problem is that the score of an item for a tag depends on *who* is asking the query, i.e., the seeker's network. One straightforward adaptation is to have one inverted list per *(tag,seeker)* pair and sort items in each list according to their score for the tag and seeker. Therefore, each item will be replicated along with its exact score in each *(tag,seeker)* inverted list. This solution is illustrated on the right-hand side of Figure 2. We refer to it as `Exact` and use it as our baseline for experiments on space overhead.

Clearly, for each tag, `Exact` would store as many inverted lists as there are distinct networks – and there could be one for each seeker in the worst case. Consider a collaborative tagging site with 100,000 users, 1,000,000 items, and 1000 distinct tags. Suppose that on an average each item receives 20 tags which are given by 5% of the users. Now, were we to maintain an inverted list for each *(tag,seeker)* pair with entries of the form *(item,score)*, the size of the index would be:
$100,000$ seekers $\times$ $1,000,000$ items $\times$ 20 tags $\times$ 5% $\times$ size of each entry. Assuming 10 bytes per entry, the size of the index is then $= 10^{12}$ bytes $= 1$ terabyte. The space requirement of 1 terabyte was obtained by using modest sizes. Actual numbers in real social tagging sites may be much higher [9]. The space requirements of `Exact` can easily become prohibitive as the network and tagging activity expand.

We now review top-k processing in the context of `Exact`.

3.2 Top-K Processing with Exact Scores

Existing top-k algorithms are directly applicable to `Exact`. Rather than describe them in detail, we give a brief overview of `NRA` (No Random Access) and `TA` (Threshold Algorithm) [6], two representative algorithms.

In `NRA`, the inverted list for each query keyword is assumed to be sorted on the exact score of items. In the first phase, the algorithm maintains a heap which contains the current candidate entries for the top-k (there could be many more than k of these). The inverted lists are scanned sequentially in parallel. When a new entry is found, it is added to the heap along with its partial (exact) score. If the item was seen before, its score in the heap entry is updated. For every heap entry, a worst-case score and a best-case score is maintained. The worst-case score is based on the assumption that the item does not appear in those lists where it is so far unseen. The best-case score assumes that the item's score in a list where it is unseen equals the bottom score of the heap (for that list). Items in the heap are sorted on their worst-case score, with ties broken using best-case scores, and subsequent ties broken arbitrarily. The algorithm stops when none of the entries outside of the top-k items examined so far has a best-case score higher than the worst-case score of the kth item in the buffer. The output of `NRA` consists of the set of top-k items in the buffer, for which we have only partial score and hence no rank information. Subsequent work (e.g., [11]) has adapted `NRA` so as to obtain exact rank information. Clearly, `NRA` could be directly

applied in the context of `Exact` by picking the lists which correspond to the current seeker and query keywords and aggregating over them as described above.

In `TA`, the inverted lists are sorted on score as for `NRA`, but random access is assumed in addition to sequential access. The algorithm accesses items in the various lists sequentially and in parallel. It maintains a heap, where items are kept sorted on their complete score. When a new entry is seen in any list under sequential access, its scores from other lists are obtained by random access. If its overall score is more than the score of the kth entry in the heap, they are swapped. Otherwise this new entry is discarded. At any stage, the bottom score (seen under sequential access) is maintained for every list and is used to compute a threshold. No unseen item can have a score higher than the threshold, so if the kth highest heap entry is greater or equal to the threshold, then the algorithm stops. The output consists of the top-k list in the buffer, including items and their scores (and hence their ranks). Unlike in `NRA`, much less of the lists will need to be explored, and the heap never needs to contain more than k items. This may result in more random accesses. The applicability of `TA` in the context of `Exact` is straightforward once the lists corresponding to the current seeker and query keywords are identified.

We now turn to describing our space-saving strategy and how top-k processing is adapted to fit it.

3.3 Computing Score Upper-Bounds

In order to save space in storing inverted lists, we factor out the seeker from each per-tag list and maintain entries of the form *(item,itemTaggers)* where *itemTaggers* are all taggers who tagged the item with the tag. In this case, every item is stored at most once in the inverted list for a given tag, as opposed to being replicated potentially once for each seeker. The question now is *which score to store with each entry*, which will in turn determine how the lists are ordered. Since scores are used for top-k pruning, it is safe to store, in each per-tag inverted list, the *maximum* score that an item could have for the tag *across all possible seekers*. Given a query keyword t_j, we have

$$\texttt{ub}(i, t_j) = max_{u \in \text{Seekers}} |\{ \quad v \mid \texttt{Tagged}(v, i, t_j)\} \; \& \\ v \in \texttt{Network}(u)|$$

In other words, the score upper-bound is the highest score an item could have for a tag. We refer to an inverted list organization based on these ideas as the `Global Upper-Bound` strategy. We will assume that the inverted lists are supplemented by the `Link` relation indexed by the seeker and the `Tagged` relation indexed by tag and item; these will support our equivalent of random access. They also support efficient computation of exact scores, given a seeker and an item.

Figure 2 shows the inverted list for tag *music* according to the `Global Upper-Bound` strategy and its counter-part in `Exact`. Notice that `Exact` may store one item multiple times across lists (e.g., item *i19* is stored with seekers *Jane*, *Amanda*, and *Leia*). In the case of `Global Upper-Bound`, an item is stored only once with its highest score ever (e.g., the score of item *i19* in `Global Upper-Bound` is higher than its score in the lists of *Leia* and *Amanda* since it corresponds to its score for *Jane*). While the per-seeker inverted lists in `Exact` are shorter than in `Global Upper-Bound`, the overall space consumption of `Exact` is expected to be much higher. However, the relative ordering of items in the inverted list

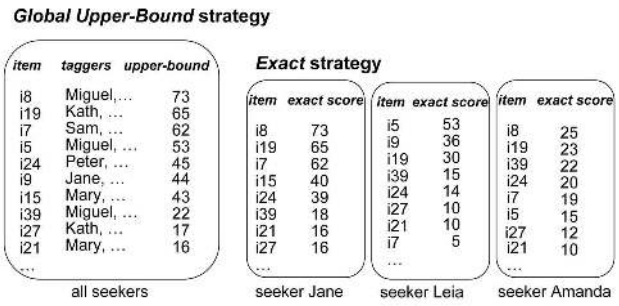


Figure 2: Inverted Lists Organization for `Global Upper-Bound` and `Exact`, for the tag *music*.

for `Global Upper-Bound` does not necessarily reflect that of any per-seeker order in `Exact` (e.g., *i19* is scored lower than *i5* in *Leia*'s list while it is scored higher in the `Global Upper-Bound` list). This may cause `Global Upper-Bound` to scan many more entries in the inverted lists than `Exact`, thereby increasing query processing time. We next describe adaptations of `NRA` and `TA` that use score bounds; their application to the bounds in `Global Upper-Bound` will serve as another baseline (for processing time) in our experiments.

3.4 Top-k Processing with Score Upper-Bounds

We assume a function that does a "local aggregation", taking a seeker and a pair *(item,itemTaggers)* from an inverted list and calculating the number of *itemTaggers* that are friends of the seeker. We also assume a function `computeExactScore`(i, u, t_j), which both retrieves the taggers of item i (for tag t_j) and counts the number of friends of u who tagged with t_j. Such a function can be implemented as a SQL aggregate query over the join of `Link` and `Tagged`.

3.4.1 NRA Generalization

Algorithm 1 shows the pseudo-code of `gNRA`. For any query, the inverted lists ($\texttt{IL}_t$) corresponding to each query keyword t are identified and are accessed sequentially in parallel, using a criterion such as round-robin. When an item is encountered in a list, we: (i) record the upper-bound of the item and (ii) compute the exact score of the item for that tag using the *itemTaggers* component of the `IL` entry. If the item has not been previously encountered, it is added to the heap along with its score. If it is already in the heap, the score is updated. Thus scores of items in the heap are partial exact scores and correspond to the notion of worst-case score of classic `NRA`. The set of bottom (i.e., last seen) bounds of the lists is used to compute best-case scores of items: for any item, its best-case score is the sum of its partial exact score and the bottom bounds of all lists where the item has not been seen. If an item is unseen anywhere, its partial exact score is 0 and its best-case score is the sum of bottom bounds over all lists. The heap is kept sorted on the worst-case score, with ties broken using best-case score, and then arbitrarily. The stopping condition is similar to that for classic `NRA`: none of the items outside the top-k of the heap has a best-case score that is more than the kth item's worst-case score. However, for classic `NRA`, it is sufficient to compare the kth item's worst-case score in the heap with the

largest best-case score of an item in the heap outside of the top-k items. In our case, this would not be sound, since the lists are only sorted on bounds, not necessarily on scores. Thus, a completely unseen item can have a best-case score higher than the largest best-case score in the heap. Thus, we compare the maximum of the largest best-case score outside of top-k in the heap and the sum of all bottom bounds with the worst-case score of the kth item in the heap, stopping when the latter is higher.

Algorithm 1 Bounds-Based `NRA` Algorithm (`gNRA`)

Require: seeker u, Query Q;
1: Open inverted lists $\texttt{IL}_t$ for each keyword $t \in Q$;
2: **while** worstcase(kth heap item) $\leq$ $max\{\text{BestcaseUnseen}, max\{\text{bestcase}(j) \mid j \in \text{heap} - \text{top-}k\}\}$ **do**
3: Get next list $\texttt{IL}_t$ using round-robin;
4: Get entry $e = (i, \texttt{ub}, \text{itemTaggers})$ in $\texttt{IL}_t$;
5: Update the bottom bound of $\texttt{IL}_t$;
6: Compute partial exact score of i for t using itemTaggers;
7: If i is not in heap add it, otherwise update its partial exact score;
8: Update best-case scores of items in heap, and re-order heap;
9: BestcaseUnseen = sum of bottom bounds over all lists;
10: **end while**
11: Return top-k set of items from heap.

At this point, we are guaranteed that the set of items in the top-k of the heap belong to the final top-k list. If the exact score (and rank) is of interest, we need to compute the exact score of items in the heap on those lists where they are not seen. We can do this by computing exact scores (our analog of Random Access) for the remaining terms of the top-k heap items. [1] We omit the pseudocode of phase 2 for brevity.

Note that during exact score computation when a cursor is moved we get the entry $e = (i, \texttt{ub}, \text{itemTaggers})$ in memory without a search, and can thus get the exact score using a local exact score computation. We will reuse the term "sequential access" (and the abbreviation SA) to refer to the combination of advancement of a cursor and local exact score computation. In our algorithm, these two always occur in tandem. We will reuse the term "random access" (RA) to refer to the calls to `computeExactScore`, which in this algorithm occur only in phase 2. The ability to quickly get the scores to be aggregated in memory allows sequential access to be much more efficient than random access, as in the classical case. We discuss this further in Section 5.

Several optimizations are possible to the basic algorithm above. Clearly, we have no need to update scores of items in the heap whose best-case is below the worst-case of the kth highest heap item, nor do we need to re-order these as the lower bounds are updated. It is also possible to check whether an element is a candidate for entry into the top-k prior to performing an exact score computation, by checking its new upper bound against the worst-case score of the current k^{th} item; this optimization can be easily incorporated into the algorithm above.

3.4.2 TA Generalization

We now present `gTA` – our adaptation of `TA` that works with score upper-bounds. Algorithm 2 shows the pseudocode. Given a query Q from a seeker u, all relevant inverted lists are identified. We access them sequentially in parallel. When an entry is seen for the first time under sequential access in a list, we compute its exact score in that list (as part of SA) and perform exact score computations on the other lists (a set of RAs). Thus, we always have complete exact scores for the items in the buffer. For each list, we remember the bottom bound seen. The threshold is the sum of the bottom bounds over all lists. The algorithm stops whenever the score of the kth item in the heap, which is kept sorted on score, is no less than the threshold. At this stage, we can output the top-k items together with their scores and hence rank them.

Algorithm 2 Bounds-Based `TA` Algorithm (`gTA`)

Require: seeker u, Query Q
1: Open inverted lists $\texttt{IL}_t$ for each keyword $t \in Q$;
2: **while** score(kth heap item) $\leq$ sum of bottom bounds over all lists **do**
3: get next list $\texttt{IL}_t$ using round-robin;
4: Let $e = (i, \texttt{ub}, \text{itemTaggers})$ be the next entry in $\texttt{IL}_t$;
5: Update the bottom bound of $\texttt{IL}_t$;
6: **if** i not in current top-k **then**
7: Use local aggregation to get exact score of i in $\texttt{IL}_t$ using itemTaggers;
8: Use `computeExactScore` to get exact score of i in other lists;
9: **if** i's overall score $>$ kth score in heap **then**
10: Swap kth item with i; keep top-k heap sorted;
11: **end if**
12: **end if**
13: **end while**
14: Output the heap as is.

As with `gNRA`, there are slight modifications that can save accesses: when we find a new item i we could iteratively retrieve scores in other lists, curtailing the iteration if we find that the best-case of i is below the exact score of the kth item in the heap.

To summarize, both variants of `Global Upper-Bound` (`gNRA` and `gTA`) differ from `Exact` in that the former needs to compute the exact score of an item for a tag and seeker at query time. Using a simple variation of the argument in [6], we can show that both `Global Upper-Bound` variants are *instance optimal* over all algorithms that use the same upper-bound based storage. In the case of `gTA`, this means that `gTA` based on a round-robin choice of cursors uses no more sequential accesses than any other "reasonable" algorithm, up to a linear factor. Reasonable here means that the algorithms can not make a call to `computeExactScore` for an item until the item has been reached under sequential access. This is the analog of the "no wild guesses" restriction of [6]. Similar statements can be made for `gNRA`: Algorithm 1 is optimal, up to a constant factor, in number of sequential accesses made, over algorithms that perform only sequential accesses. If we consider the optimization where exact score computations are done only for items that have a best-case score above the current kth highest-score, we find that it is optimal in terms of the number of exact score computations.

However, any of these optimality statements only justify the use of these query processing algorithms once a storage structure is fixed; they do not justify the storage structures themselves. The accuracy of upper-bounds in the inverted list is clearly the key factor in the efficiency of top-k prun-

[1] Thus, `gNRA` in our setting is really "generalized Not many Random Accesses".

ing. The finer the upper-bound, the closer it is to the item's exact score and the faster an item can be pruned. Therefore, we need to explore further optimizations of our inverted list storage strategies. The clustering-based approaches introduced in the next section work by identifying upper-bound based inverted lists for a *(query,seeker)* pair and then applying either `gNRA` or `gTA`. They will differ on which lists are identified and on how the clusters are formed.

4. CLUSTERING & QUERY PROCESSING

As discussed in Section 3, it is intuitively clear that the greater the distance between the score upper-bound and the exact score of an item for a tag, the more processing may be required to return the top k results for a given *(seeker,query)* pair. The aim of this section is to describe methods which *reduce the distance between exact scores and upper-bounds* by refining upper-bounds. The core idea is to *cluster users* into groups and compute score upper-bounds within each group. The intuition is that by making a cluster contain users whose *behavior* is similar, the exact scores of users in the cluster will be very close to their upper-bound score. The remaining question is thus: *which users should the clustering algorithm group together to achieve that goal.*

4.1 Clustering Seekers

Since the score of an item depends on the network of taggers, a natural approach is to cluster the seekers based on similarity in their scores. Given any clustering of seekers, we form an inverted list $\texttt{IL}_{t,C}$ for every tag t and cluster C, containing all items tagged with t by a tagger in $\bigcup_{u \in C} \texttt{Network}(u)$, with the score of an item being the *maximum score over all seekers in the cluster*. That is, an item i in the list gets score $\max_{u \in C} \texttt{score}_t(i, u)$. Query processing for $Q = t_1 \dots t_n$ and seeker u proceeds by finding the cluster $C(u)$ containing u and then performing aggregation (using one of the algorithms in Section 3) over the collection of inverted lists $\texttt{IL}_{t_i,C(u)}$.

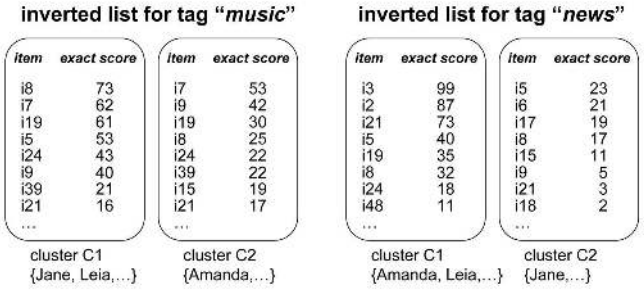


Figure 3: Example of `Cluster-Seekers`

`Global Upper-Bound` is a special case of `Cluster-Seekers` where all seekers fall into the same cluster and the same cluster is used for all tags. That is not the case in general with `Cluster-Seekers` as illustrated in Figure 3 where *Jane* and *Leia* fall into the same cluster C_1 for the tag *music*, but into different clusters for the tag *news*. Upper-bounds within each list are computed for members of the associated cluster, which makes the bounds necessarily tighter. For example, item *i7* has a tighter upper-bound in cluster C_2 for tag *music* than in `Global Upper-Bound` in Figure 2. It is necessary to compute seekers' clusters on a per-tag basis in order to reflect (i) the overlap in networks among different seekers *and* (ii) the overlap in tagging actions of taggers in clusters. Indeed, if we generated a set of clusters which are agnostic to the tags, we could end up creating a cluster containing *Jane*, *Leia* and *Amanda* (since they share some taggers in their networks). The score upper-bounds for such a cluster would be coarser (e.g., *i19* would have a score upper-bound equal to 65 for tag *news* while it is equal to 30 in C_2!).

One can easily show that, for single keyword queries, as we refine the clustering, the upper bounds can only get tighter, and hence the processing time of the generalized threshold algorithms of the previous section can only go down, regardless of the seeker. In particular, `Exact` is optimal over all algorithms that use `Cluster-Seekers`, for any clustering, assuming single-keyword queries.

For multi-keyword queries, this is not the case. Consider a seeker u and a top-1 query $Q = t_1, t_2$. Assume that, for this seeker, the lists of scores for the two keywords are $IL_{t_1,u} = (x{:}10, i_1{:}2, i_2{:}1.5, \dots)$, where the ... is a long tail of scores below 1.5; $IL_{t_2,u} = (i'_1{:}20, i'_2{:}19, \dots, i'_{1000}{:}19, \dots, x{:}15)$. Suppose a clustering puts this seeker in her own cluster, thus making the bounds exact for her! Under `gNRA`, we would have to keep making sequential accesses until we reach the entry x:15 in the second list. However, it is possible that in `Global Upper-Bound` (the coarsest possible clustering), the global inverted lists for the two keywords are: $IL_{t_1} = (x{:}10, i_1{:}2, i_2{:}1.5, \dots)$; $IL_{t_2} = (x{:}22, i'_1{:}20, i'_2{:}19, \dots, i'_{1000}{:}19)$. The entry x might have a much coarser (i.e., higher) bound in `Global Upper-Bound` and hence may bubble to the top of the second list. Now, `gNRA`, after one round-robin, gets the full exact score of x (still 25 for u). After this the algorithm will stop, since this score is higher than the sum of all bottom scores. This example shows that *an ideal clustering would take into account both the scores and the ordering.*

How do we cluster seekers? Ideally, we would find clusters that are *optimal* for the running time of one of our algorithms in Section 3. We could look at the worst-case running time over all users (for, say, a particular tag or set of tags). For simplicity, we consider the case of single keyword queries, where the distinction between `gNRA` and `gTA` will not be important. Consider a seeker u asking a query for a single keyword t, where we want to return the top k answers. For a fixed data set D, set of seeker clusters $C_1 \dots C_n$, let $\text{CompTime}(C_1 \dots C_n, D, t, u, k)$ be the number of sequential accesses made in `gNRA` or `gTA` for u while doing top-k processing for query t over D, where processing is done using the inverted lists $\texttt{IL}_{t,C_i}$ as discussed above. Let $\text{WCompTime}(C_1 \dots C_n, D, t, k)$ be

$$\max_{u \in \texttt{Seekers}} \text{CompTime}(C_1 \dots C_n, \text{D}, \text{t}, u, k)$$

Given (n, D, t, k), one would like to find $C_1 \dots C_n$ that minimizes $\text{WCompTime}(C_1 \dots C_n, D, t, k)$.

It is well-known that finding clusters of points that minimize the diameter within a metric space is NP-hard, even for metrics in 2-dimensions [8]. This does not immediately imply that clustering is hard in our setting, since our objective function $\text{WCompTime}(C_1 \dots C_n, D, t, k)$ does not correspond to the diameter in a metric. Indeed, the behavior of `gNRA` and `gTA` depends not on pairwise interactions of the items tagged by users in the cluster, but the relationship between all the ordered lists. Nevertheless, we can show:

THEOREM 1. *The problem of finding $C_1 \dots C_n$ that minimizes WCompTime($C_1 \dots C_n, D, t, k$) (given D, t, k) is NP-hard, even for $n = 2$ and $k = 1$.* ∎

The proof is by reduction from the *independent task scheduling problem* which is known to be NP-hard [8] even for 2 processors. The rough idea of the reduction is that tasks are mapped to items tagged by taggers in distinct networks, with the number of users tagging an item corresponding to the time delay induced by the task. A clustering will yield inverted lists that partition the items, hence yielding a partition of tasks.

Similar results can be shown for the *average case computing time* for a cluster. We state here only one result, which shows that one cannot easily find the clusters that minimize the size of the inverted lists for most users. Given a data set D, a set of seeker clusters $C_1 \dots C_n$, a tag t, and a seeker u, we let SZ($C_1 \dots C_n$,D,t,u) be the size of the inverted list $\mathtt{IL}_{t,C(u)}$, and let AVGSZ($C_1 \dots C_n, D, t$) = $\mathrm{AVG}_{u \in \mathtt{Seekers}}$ SZ($C_1 \dots C_n, D, t, u$). We then have:

THEOREM 2. *Finding the clusters that minimize AVGSZ($C_1 \dots C_n$) is NP-hard, even for $n = 2$.* ∎

The proof is by reduction from the *minimum sum of squares problem* [8]. Again a collection of items is used that are tagged by distinct networks, with the networks associated with many users.

Given the above results, we must rely on heuristic methods to find clusters of seekers. One natural approach is based on overlap of the seekers' networks. The intuition is that, given two seekers u and u', the higher the number of common taggers in their networks, the higher the chance that the score of an item for those networks be similar. However, two taggers may have different tagging behaviour for different tags. Therefore, we propose to compute per-tag network overlap between seekers.

Given the set of all seekers in `Seekers`, we can construct a graph where nodes are seekers and an edge between two seekers u and u' is created iff $|\mathtt{Network}(u,t) \cap \mathtt{Network}(u',t)| \geq thres$, where $\mathtt{Network}(u,t)$ is the set of users in $\mathtt{Network}(u)$ who tagged at least one item with tag t. The threshold $thres$ is application-dependent. Once the graph is instantiated, we apply off-the-shelf graph clustering algorithms in Section 5 to generate clusters. We report the space/time performance of `Cluster-Seekers` in Section 5.4.

4.2 Evaluating and Tuning Clusters

The `Cluster-Seekers` approach depends on finding good clusterings, which in turn depends on being able to predict the performance of a clustering. Clearly, we cannot test a clustering on all keyword combinations for all users, and for all values of k. We begin with a measure comparing the orderings produced by the clustering with the exact orderings for a particular user; we will average this over users. Although there are many metrics on lists considered in the literature (see, e.g. [5]), we need one that is both correlated with the performance of our algorithms and simple to compute. We discuss here one such measure, a variant of Normalized Discounted Cumulative Gain (NDCG, [12]).

Fix a tag t and seeker u, and let L_{ideal} be the ranked list of items for u and t, ranked by exact scores, with M_{ideal} being the length of L_{ideal}. Let L_{approx} be the ranked list of items in a cluster, ranked by cluster upper-bounds. The underlying domain of L_{approx} is a superset of the domain of L_{ideal}. Let $D(i)$ be the maximum over $j \leq i$ of the position of the j^{th} item in L_{ideal} within L_{approx}. The *Gain Vector* for u, t is a vector of length M_{ideal} such that $G(i) = i/D(i)$. That is, $D(i)$ represents the delay in getting to the top i items of L_{ideal}, hence a low gain $G(i)$ represents a high delay for using this list, and vice versa. $G(i)$ is at most one, since we must wait at least i to reach i items.

Intuitively, $G(i)$ represents the quality of the clustered list if we are looking for the seeker's top i items. For top-k multi-keyword queries, the highest scoring item may be significantly further down on an individual list than k, and we do not know before query time how far down it may come. We will thus sum the quantity $G(i)$ over i, discounting higher values of i, since it is less likely that the tail of the seeker's list will be relevant.

Given a "discount factor" b, let $G'(i) = G(i)/log_b(i)$ and $l'(i) = 1/ilog_b(i)$ for $i > b$, and let $G'(i) = G(i)$ and $l'(i) = 1/i$ otherwise. (We use $b = 2$ in our experiments.) We define the *Normalized Discounted Cumulative Gain* (NDCG) of the clustered list with respect to u as $\frac{\Sigma_{i<D}(G'(i))}{\Sigma_{i<D} l'(i))}$. The denominator is an additional normalization factor that guarantees that the entire quantity is at most one.

NDCG measures the quality of a clustered list for a given seeker and keyword. A value of 1 is the optimum, which is realized only by the ideal list, while values close to 0 indicate a list that is far from ideal. The quality over all seekers can be estimated by averaging over a randomly selected collection of seekers. We will see in Section 5 that the run-time performance of `Cluster-Seekers` is correlated with NDCG. The performance of a multi-keyword query Q is a function of the "aggregate NDCG of Q", with the aggregation ranging over a sample of seekers and over the keywords in Q.

The NDCG can be used to compare two clusterings – for example, those done via different clustering algorithms, or different parameters within a clustering algorithm. It can also be used to decide whether increasing the number of clusters will significantly impact performance. Since the NDCG is a per-keyword quantity, it can be calculated offline.

4.3 Clustering Taggers

Another clustering alternative is to organize taggers into different groups which reflect overlap in their tagging behavior. We refer to this strategy as `Cluster-Taggers`. That is, for each tag t we partition the taggers into clusters. We again form inverted lists on a per-cluster, per-tag basis, where an item i in the inverted list for cluster C and tag t gets the score:

$$\max_{u \in \mathtt{Seekers}} |\mathtt{Network}(u) \cap C \cap \{v \mid \mathtt{Tagged}(v,i,t)\}|,$$

i.e., the maximum number of taggers in cluster C who are linked to u and tagged item i with tag t, over all seekers u. To process a query $Q = t_1 \dots t_n$ for seeker u, we find the set of clusters of the taggers in `Network` (u), and then perform an aggregation over the inverted lists associated with all *(tag,cluster)* pairs. Members of a seeker's network may fall into multiple clusters for the same tag, thereby requiring us to process more lists for each tag (as opposed to one list per tag in the case of clustering seekers).

How do we cluster taggers? We instantiate a graph where nodes are taggers and there exists an edge between two nodes v and v' iff: $|\mathtt{Items}(v,t) \cap \mathtt{Items}(v',t)| \geq thres$ where t is any tag. Again, $thres$ depends on the application.

We now summarize the differences between clustering seekers based on network overlap (`Cluster-Seekers`) and clustering taggers based on overlap in tagging behavior (`Cluster-Taggers`). At query time, `Cluster-Seekers` identifies one inverted list per *(tag,seeker)* pair since a seeker always falls into a single cluster for a tag. In `Cluster-Taggers` there are potentially multiple inverted lists per *(tag,seeker)* pair, given that a seeker will generally have multiple taggers in his network which may fall into different clusters.

Unlike `Cluster-Seekers`, `Cluster-Taggers` does not replicate tagging actions over multiple inverted lists. In fact, we will show that there is no significant penalty in space of `Cluster-Taggers` over `Global Upper-Bound`. Space consumption of clustering is explored in Section 5.

As for processing time, while `Cluster-Seekers` benefits all seekers, `Cluster-Taggers` does not. Indeed, we find that `Cluster-Taggers` can hinder seekers that are associated with many tagger clusters and hence many inverted lists. Still, we show that there is a *significant* percentage of seekers that can benefit from `Cluster-Taggers` and that *this population can be identified in advance*. `Cluster-Taggers` also has advantages for maintenance under updates; while `Cluster-Seekers` requires multiple exact score computations and updates to maintain as new tagging events occur, `Cluster-Taggers` requires only a single exact score computation and a single update per tag. This is because in `Cluster-Taggers`, items are not replicated across clusters.

5. EXPERIMENTAL EVALUATION

5.1 Implementation

We implement the `gNRA` and `gTA` algorithms in Java on top of an Oracle 10g relational database. Our experiments are executed on a MacBook Pro with a 2.16GHz Intel Core 2 Duo CPU and 1GB of RAM, running MacOS X v10.4. The database server is running on a 64-bit dual processor Intel Xeon 2.13GHz CPU with 4GB or RAM, running RedHat Enterprise Linux AS4.

Our schema consists of the following relations:
`TaggingActions(itemId,taggerId,tag)` stores raw tagging actions.
`Link(tag,seekerId,taggerId)` encodes the `Link` relation between seekers and taggers, on a per-tag basis. The network of a seeker is a union of all taggers associated to it.
`InvertedList(tag,clusterId,itemId,ub)` stores inverted lists of items per tag, per cluster. A tree index is built on (tag, ub) to support ordered access. This table also stores per-tag inverted lists for `Global Upper-Bound`.
`SeekerClusterMap(seekerId,clusterId,tag)` stores the assignment of seekers to clusters.
`TaggerClusterMap(taggerId,clusterId,tag)` stores the result of clustering `Taggers`.

Given a seeker u, and a tag t, an SA is implemented as moving a cursor over the result of the query:

```
Select IL.itemId, IL.ub, count(*) as ''score''
From   InvertedList IL, TaggingAction T, Link L
Where  L.seekerId = :u And T.tag = :t
And    L.taggerId = T.taggerId And T.tag = L.tag
And    T.tag = IL.tag And T.itemId = IL.itemId
Group by IL.itemId, IL.ub
Order by IL.ub descending
```

Appropriate indexes are built on the join columns to ensure efficient access. The role of aggregation in this query is to compute partial exact scores of items with respect to the current "inverted list". An RA, i.e. a calculation of `computeExactScore` on a single item i, is given as follows:

```
Select count(*) as ''score''
From   TaggingAction T, Link L
Where  L.seekerId= :u And T.tag= :t And T.itemId= :i
And    L.taggerId = T.taggerId And L.tag = T.tag
```

Both queries are for `Global Upper-Bound`, and are augmented by a join with `SeekerClusterMap` for `Cluster-Seekers` and by a join with `TaggerClusterMap` for `Cluster-Taggers`.

We use a SQL-based implementation for convenience only. The exact difference in cost between SAs and RAs, and hence the overall performance time of the algorithms, may vary significantly from a native implementation of the algorithms using inverted lists. However, we will quantify the query execution time of all our algorithms via the number of SAs and RAs. This will thus suffice to give a picture of the relative performance of our algorithms both in comparison to `Exact` and to each other.

We found that, as is the case with traditional top-k algorithms, the RAs in our implementation are significantly more expensive than SAs. The relative cost varies slightly depending on the tag, but a single RA is about 10 times more expensive than a single SA.

We use `Graclus` [3], an efficient clustering implementation over *undirected weighted graphs*. `Graclus` provides two clustering methods. Ratio Association (ASC) maximizes edge density within each cluster, while Normalized Cut (NCT) minimizes the sum of weights on edges between clusters [3]. More formally, given two sets of nodes $\mathcal{V}_i$ and $\mathcal{V}_j$, we denote by $links(\mathcal{V}_i, \mathcal{V}_j)$ the sum of edge weights between nodes in $\mathcal{V}_i$ and nodes in $\mathcal{V}_j$. We denote by $degree(\mathcal{V})$ the sum of weights on edges incident on the nodes in $\mathcal{V}$. The objective of ASC is: `maximize` $\sum_{i=1}^{n} \frac{|links(\mathcal{V}_i,\mathcal{V}_i)|}{|\mathcal{V}_i|}$ The objective of NCT is: `minimize` $\sum_{i=1}^{n} \frac{|links(\mathcal{V}_i,\mathcal{V}\setminus\mathcal{V}_i)|}{degree(\mathcal{V}_i)}$.

5.2 Data and Evaluation Methods

We use del.icio.us datasets for our experimental evaluation. del.icio.us is a collaborative tagging site where users bookmark URLs (to which we refer as items) and optionally annotate them with tags. The dataset is very sparse and follows a long tail distribution [13, 9]: most items are tagged by only a handful of users, and many tags are only used by a few users. We were given one month of data by the del.icio.us team. In order to reduce the size of the dataset for easier processing by Oracle 10g, a commercial RDBMS, we removed all items that were tagged by fewer than 10 distinct users. Additionally, we removed tagging actions that include uncommon tags: only tags used by at least 4 distinct users are included in our dataset. This dramatically reduced the cardinality of the group by queries used to compute inverted lists with upper bounds (see Subsection 5.1). The impact on the run-time results of our top-k algorithms is limited, since only items that were "unpopular" over all tags and all users were removed. As a result, the dataset was reduced to 27% of its original size, and contains 116,177 distinct users who tagged 175,691 distinct items using 903 distinct tags, for a total of 2,322,458 tagging actions. For the purposes of our evaluation we focus on users who contributed at least

tag	|Net|	avg|Net|(v)	max|Net|(v)
Software	25545	8	607
Programming	21853	21	983
Tutorial	16895	23	1068
Reference	24697	34	1098

Table 1: Characteristics of `Network` for four tags.

one tagging action to the cleaned dataset.

We choose 4 tags (*software*, *programming*, *tutorial*, and *reference*) from the 20 most popular. Popularity of a tag is measured by the total number of tagging actions involving it: the most popular tag has been used about 100,000 times, while the 20th most popular was used about 34,000 times. We evaluate the performance of our methods over 6 queries of varying lengths, to which we refer by the first letters of each tag (e.g. *SP* for *software programming*). We chose these four tags because they are thematically related and may be meaningfully combined in a query.

del.icio.us has an explicit notion of friendship, but users may have various semantics for it. Since we were interested in networks that reflect affinities in item preference, in our experiments we use a network derived from the tagging data via *common interest*. There is a link between a seeker and a tagger if they tagged at least *two* items in common with the same tag. Table 1 lists the number of users per tag ($|Net|$), as well as the average and maximum cardinalities of `Link`, i.e., the size of a seeker's network.

We use the following sampling methodology to select users for our performance evaluation. For each tag and for each seeker we compute the total number of tagging actions that are relevant to that seeker (i.e., the total number of tagging actions by all taggers linked to the seeker), and rank seekers on this value. We notice that the top 25% of the seekers together correspond to 75%-80% of all tagging actions for the four tags in our experiments. For each query we identify three mutually exclusive groups of seekers: *Seekers-25* are in the top 25% of ranks for each query keyword, *Seekers-50* are in the top 50% of ranks for each query keyword, but not in the top 25%, *Seekers-100* are the rest. For each query we draw 10 seekers uniformly at random from each group, for a total of 30 seekers per query. This methodology allows us to capture the variation in performance for different types of seekers: popular tags correspond to many more items for `Seekers-25` than for `Seekers-100`.

We evaluate the performance of our algorithms with respect to two metrics. *Space overhead* is quantified by the number of entries in the inverted lists. *Query execution time* is expressed by the number of sequential and random accesses (SAs and RAs). The raw number of accesses varies significantly between seekers even when the query is fixed. Hence, we focus on *relative improvement* obtained by `gNRA` and `gTA` compared to `Global Upper-Bound`. Unless otherwise stated, we report average percent improvement over the baseline for 30 seekers per query, with separate averages given for *Seekers-25*, *Seekers-50* and *Seekers-100*. We use truncated mean, and remove the minimum and maximum values before computing the average.

5.3 Performance of Global Upper-Bound

We start with some general comments about how the number of SAs and RAs increases with k, for both `gNRA` and `gTA`. The qualitative behavior depends on the characteristics of the seeker's network. Consider the query *software programming* for 3 selected users in Figure 4. For a fixed user, `gNRA` and `gTA` exhibit the same trend in both SAs and RAs. How the number of accesses increases with k is a function of the distribution of exact scores. For example, the number of SAs for `gNRA` for seeker u_2 increases dramatically for $k = 40$. When looking at the distribution of exact scores for this seeker we notice that the items can be classified into 3 categories with respect to their score: the first 36 items score higher than 3, followed by 60 items with a score of 2. The remaining 405 items (79%), have a score of 1, and constitute the tail of the distribution for seeker u_2. The spike in accesses occurs when k becomes high enough that the long tail needs to be explored.

query	Seekers-25		Seekers-50		Seekers-100	
	GUB	Exact	GUB	Exact	GUB	Exact
SP	1674	52	12920	134	18036	61
PR	479	13	3923	87	12982	61
TR	1262	14	4813	92	18476	121
SPT	938	78	4107	112	17985	195
SPR	1495	67	8972	194	14976	131
SPTR	907	119	2229	119	10986	189

Table 2: `gNRA` (SA) in `Global Upper-Bound` and `Exact`.

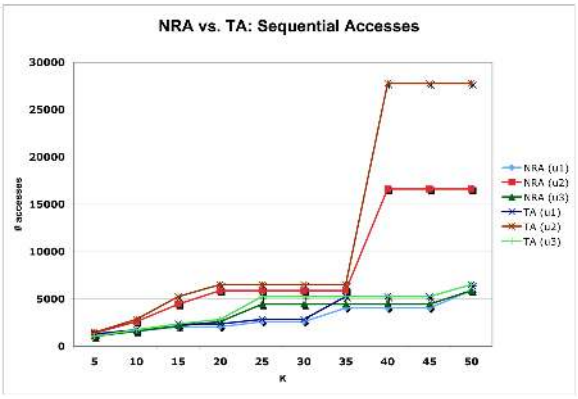
Space overhead of `Global Upper-Bound` is presented in Figure 5; compared to `Exact`, `Global Upper-Bound` achieves savings of over two orders of magnitude. However, as argued in Section 3, the `Global Upper-Bound` strategy, while optimal with respect to space overhead, may suffer from high query execution time. Table 2 compares the number of SAs for `gNRA` under `Global Upper-Bound` to the number of SAs for `Exact`. We observe that `Global Upper-Bound` underperforms `Exact` by up to two orders of magnitude, and that users in *Seekers-100* are at a particular disadvantage. `gTA` under `Global Upper-Bound` shows a similar trend.

5.4 Clustering Seekers

We experiment with 3 clustering algorithms: ASC, NCT (Section 5.1) and a random clustering for reference, RND, which assigns seekers to random clusters. We cluster over the *common-interest network* of seekers: there is an edge between two nodes u_1 and u_2 if these users tagged at least one item in common. This graph is undirected, and edges are weighted by the number of items tagged in common.

Space. Figure 5 (left) summarizes the space overhead of clustering as the cluster budget varies from 10 to 500 clusters. `Global Upper-Bound` has lowest overhead, with 74,181 total inverted list entries, while `Exact` has 62,973,876 entries. Space overhead of NCT ranges between 533,346 rows for 10 clusters and 4,284,854 rows for 500 clusters; ASC stores between 472,401 and 6,794,890 rows; while RND stores between 643,994 and 14,543,547 rows. ASC and NCT both achieve an order of magnitude improvement in space overhead over `Exact`. At this stage, we discard RND due to relatively poor space utilization, fix the number of clusters at 200, and continue our experiments with ASC (4,448,717 rows) and NCT (2,984,377 rows).

Time. Tables 3 and 4 quantify the performance of `gNRA` and `gTA` with `Cluster-Seekers` when NCT and ASC are used for clustering. We list improvement in the number of sequential accesses (# SA) and in the **total** number of accesses, which is simply # SA + # RA. We observe that both `gNRA` and `gTA` with `Cluster-Seekers` significantly outperform `Global`



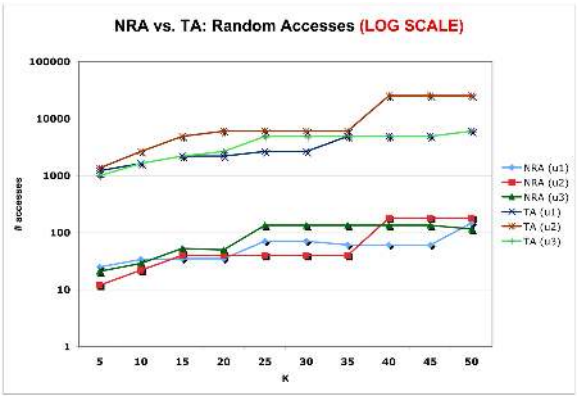
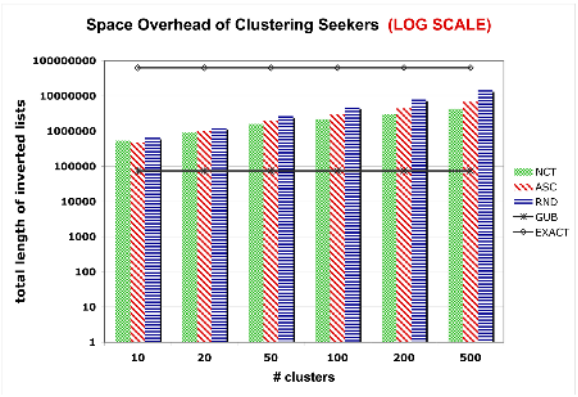


Figure 4: Performance of gNRA and gTA as K varies.



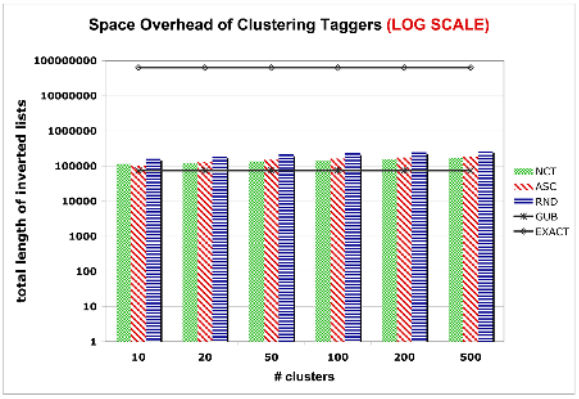


Figure 5: Space overhead of Cluster-Seekers and Cluster-Taggers.

Upper-Bound with both types of clustering. Consider the average improvement in the total number of accesses achieved by gNRA. With NCT, gNRA makes 38-72% fewer total accesses compared to Global Upper-Bound, and with ASC the total number of accesses is improved by 67-87%. We observe a similar trend for gTA: NCT improves average total accesses by 42-69%, while ASC improves by 72-83%. Further, we observe that ASC outperforms NCT on both sequential and total accesses in all cases for gTA, and in all cases except one in gNRA, query *TR* for *Seekers-50*, where NCT is better by 2%. Finally, note that in most cases % improvement over Global Upper-Bound is highest for *Seekers-100*, followed by *Seekers-50*. However, this trend needs to be related to the findings in Table 2: for *Seekers-100* Global Upper-Bound performs worst compared to Exact, and so there is significant room for improvement. Also note that improvement in # SA is similar to improvement in the total number of accesses. This is because performance of gNRA is heavily dominated by sequential accesses, while in gTA, # RA is bounded by # SA $*(n-1)$ for a query of length n.

In Section 4.2 we described a variant of NDCG, a clustering quality metric that is correlated with the run-time performance of our algorithms. NDCG captures the run-time degradation compared to Exact, and we now demonstrate the correlation between NDCG and run-time performance of gNRA. In Tables 5 and 7 we show that NDCG is predictive of % degradation over Exact for Global Upper-Bound, NCT and ASC, for all groups of users in our experiments. For gNRA, SAs dominate the run time by a large margin so we use SA to quantify run time in this experiment. Higher values of NDCG correlate with lower values of % degradation in

	Seekers-25					
	NDCG			% degr. over Exact		
query	GUB	NCT	ASC	GUB	NCT	ASC
SP	0.0575	0.0841	0.2232	2594	1128	306
TR	0.1155	0.1653	0.2602	796	382	180
PR	0.0796	0.1215	0.1745	2034	737	354
SPT	0.0936	0.1366	0.2963	963	569	123
SPR	0.0737	0.1283	0.2170	1571	626	286
SPTR	0.1098	0.1674	0.2883	739	362	135

Table 5: Using NDCG to predict run-time performance of clustering for *Seekers-25*.

SAs compared to Exact. Global Upper-Bound has consistently lower NDCG, leading to poor query-execution times, while ASC has highest values of NDCG, and best run-time performance compared to other methods. A similar trend holds for gTA. It also holds when the total number of accesses rather than the number of SAs is used to measure run-time. Recall that NCT outperformed ASC for *Seekers-50* for query *TR* (see Table 4). NDCG captures this, assigning the highest value to NCT for this query and user sample. NDCG does mis-predict the relative performance of ASC and NCT in a single case, for the query *TR* on *Seekers-100*.

We conclude with a scatter-plot (Figure 6) that presents the correlation between NDCG and the number of SAs for *Seekers-25*, for all clustering methods and all queries.

5.5 Clustering Taggers

We cluster taggers using a variation of the underlying link relation in our experimental network: there is an edge between two taggers if they tagged at least one item in common

query	NCT (% improvement over GUB)						ASC (% improvement over GUB)					
	Seekers-25		Seekers-50		Seekers-100		Seekers-25		Seekers-50		Seekers-100	
	SA	Total	SA	Total	SA	Total	SA	Total	SA	Total	SA	Total
SP	35	34	76	75	78	77	83	79	85	84	89	89
TR	54	53	78	77	78	80	80	72	76	75	85	84
PR	37	35	75	74	70	70	63	59	82	81	82	82
SPT	31	28	41	38	73	71	74	66	80	76	85	83
SPR	52	46	58	55	73	71	73	67	81	78	89	87
SPTR	40	34	49	45	64	60	68	56	78	70	82	77
Average	42	38	63	61	73	72	74	67	80	77	85	84

Table 3: Performance of `gNRA` over `Cluster-Seekers` with 200 clusters.

query	NCT (% improvement over GUB)						ASC (% improvement over GUB)					
	Seekers-25		Seekers-50		Seekers-100		Seekers-25		Seekers-50		Seekers-100	
	SA	Total	SA	Total	SA	Total	SA	Total	SA	Total	SA	Total
SP	36	34	66	65	73	73	80	80	82	81	87	87
TR	54	54	72	72	76	75	72	72	77	77	82	82
PR	38	37	74	74	68	67	64	64	81	81	79	78
SPT	34	35	46	45	74	73	72	73	81	80	84	84
SPR	53	52	57	56	67	66	74	74	78	77	85	85
SPTR	41	42	50	49	63	61	68	67	77	78	82	81
Average	43	42	61	60	70	69	72	72	79	79	83	83

Table 4: Performance of `gTA` over `Cluster-Seekers` with 200 clusters.

query	`Clus. Seekers`		`Clus. Taggers`	
	# SA	Total	# SA	Total
SP	82	82	97	97
TR	78	78	94	94
PR	82	82	97	96
SPT	83	83	97	97
SPR	86	86	95	95
SPTR	83	83	96	96

Table 6: % improvement of `Cluster-Seekers` and `Cluster-Taggers` over `Global Upper-Bound`.

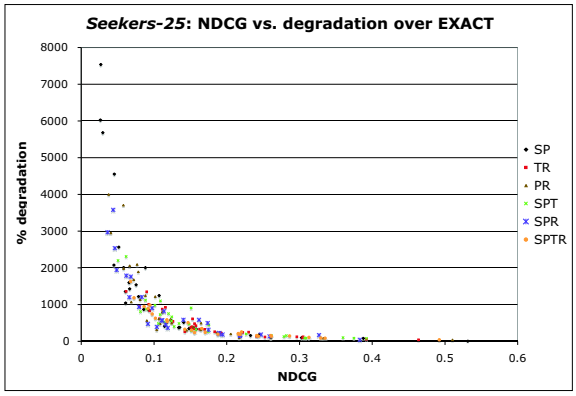


Figure 6: Correlation between NDCG and #sequential accesses for *Seekers-25*.

with a given tag. Edges are weighted by the number of items tagged in common.

Space. Figure 5(right) presents the space overhead of `Cluster-Taggers` on a logarithmic scale. As expected, space overhead of this method is significantly lower than that of `Exact` and of `Cluster-Seekers`. Space overhead of NCT ranges from 115,168 rows for 10 clusters to 167,141 rows for 500 clusters; the overhead of ASC is between 100,922 and 180,881 rows; RND consumes between 160,432 and 259,216 rows. These numbers are all comparable to the optimal space consumption of `Global Upper-Bound`: 74,181 entries, which is explained by the lack of duplication of entries in the lists.

Time. In the best case, given a keyword, all taggers relevant to a seeker will reside in a single cluster; then, only one inverted list will be processed at query time for that keyword. In the worst case, all taggers in the seeker's network will reside in separate clusters. For 200 clusters and tag *reference*, there are 34 taggers per seeker, on average (Table 1).

`Cluster-Taggers` has low space overhead; here, query execution time is the bottleneck. Since we found ASC to perform well in terms of time in other contexts (e.g. see Section 5.4), we now focus our attention on ASC with 200 clusters.

Even under the most effective clustering, a seeker may still be mapped to many clusters. An extreme case in our dataset is a seeker who mapped to 80 clusters for the two-keyword query *SP* (this seeker has 323 taggers). As a result, the number of SAs of `gNRA` increased 26 times compared to `Global Upper-Bound`, clearly an unacceptable performance. We observed empirically that `gNRA` with `Cluster-Taggers` outperforms `Global Upper-Bound` when at most $3 * queryLength$ clusters are identified for the seeker. Therefore we propose to use `Cluster-Taggers` for a subset of the seekers – those who map to at most $3*queryLength$ clusters. In our dataset between 46-68% of seekers map to at most 3 clusters per tag.

Table 6 compares the run-time performance of `Cluster-Taggers` and `Cluster-Seekers` to `Global Upper-Bound`. We used ASC with 200 clusters for both `Cluster-Taggers` and `Cluster-Seekers`. We used a different sampling methodology for this experiment. For each query, we identified the set of seekers who map to at most 3 clusters for each keyword and sampled 10 seekers uniformly at random from that set. `Cluster-Taggers` outperforms `Cluster-Seekers` for all queries in our experiments, and achieves 94-97% improvement over `Global Upper-Bound` for this class of seekers.

6. RELATED WORK

Top-K Processing: Top-k algorithms aim to reduce the amount of processing required to compute the top-ranked answers, and have been used in the relational [2], XML [15], and other settings. The core ideas of these algorithms are overviewed in [6, 4]. A common assumption is that scores are pre-computed and used to maintain dynamic thresholds

query	Seekers-50						Seekers-100					
	NDCG			% degr. over Exact			NDCG			% degr. over Exact		
	GUB	NCT	ASC	GUB	NCT	ASC	GUB	NCT	ASC	GUB	NCT	ASC
SP	0.0270	0.0735	0.1088	8605	1893	1229	0.0100	0.0802	0.1080	26238	5388	2707
TR	0.0435	0.1206	0.1084	3935	728	813	0.0176	0.0728	0.0895	31835	4198	3609
PR	0.0391	0.0894	0.1093	4268	945	753	0.0183	0.0933	0.0430	19179	5051	3562
SPT	0.0459	0.0766	0.1436	3221	1735	537	0.0261	0.0857	0.1023	7597	1664	977
SPR	0.0381	0.0846	0.1319	4039	1407	487	0.0277	0.0833	0.1034	9293	1927	892
SPTR	0.0670	0.1099	0.1847	1684	766	154	0.0421	0.0914	0.1236	4103	1376	606

Table 7: Using NDCG to predict run-time performance of clustering for *Seekers-50* and *Seekers-100*.

during query processing, in order to efficiently prune low-scoring answers. Even in work where the underlying query model is distributed [16], or where the aggregation computation is expensive [10], this assumption of pre-computation remains in place. In our work, we extend Fagin-style algorithms to process score *upper-bounds* – since pre-computed scores for each individual seeker are too expensive to store (given that they depend on a seeker's network) – and we explore clustering as a way to refine upper-bounds and reduce the size of the inverted lists.

Socially Influenced Search: Although the potential of using social ties to improve search has been recognized for some time, this is still subject of ongoing work. Current research has focused on judging the impact of various notions of user affinity and socially-influenced scoring functions on search quality [17, 18, 21, 14]. In contrast, our work develops indexing and query evaluation methods which apply to a wide class of scoring functions and networks.

Collaborative Tagging Sites: Social graphs obey a "power-law" distribution [19]. This applies to online tagging, where a large number of items are tagged by a handful of users and a large fraction of tags are used a small number of times [9, 13]. We believe that improving search results on tagging sites is a good step towards encouraging user participation and reducing data sparsity.

7. CONCLUSION

We presented network-aware search, a first attempt to incorporate social behavior into searching content in collaborative tagging sites. These sites present a unique opportunity to account for explicit and implicit social ties when scoring query answers. We defined a top-k processing model which relies on a family of scoring functions that compute item popularity among a network of users. We extended traditional top-k algorithms, and explored clustering users as a way to achieve a balance between processing time and space consumption. Our solution improves on the current state of the art in search in collaborative tagging sites, but also reaches beyond social tagging sites, and constitutes a first attempt at incorporating a social dimension into web search. One immediate improvement is the verification of our results on other collaborative tagging sites. Another is to explore alternative user clusterings, e.g. a combination of `Cluster-Seekers` and `Cluster-Taggers`.

8. REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[2] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *SIGMOD*, 1997.

[3] I. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors: A multilevel approach. *IEEE Trans. Pattern Anal. Mach. Intel.*, 29(11), 2007.

[4] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 32(2), 2002.

[5] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM Journ. Discr. Math.*, 17(1), 2003.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4), 2003.

[7] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM TOIS*, 15(1), 1997.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guite to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[9] S. A. Golder and B. A. Huberman. The structure of collaborative tagging systems. *Information Dynamics Lab, HP Labs*, 2006.

[10] S.-W. Hwang and K. C.-C. Chang. Probe minimization by schedule optimization: Supporting top-k queries with expensive predicates. *IEEE TKDE*, 19(5), 2007.

[11] I. Ilyas, W. Aref, and A. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, 2002.

[12] K. Järvelin and K. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM TOIS*, 20(4), 2002.

[13] M. E. Kipp and D. G. Campbell. Patterns and inconsistencies in collaborative tagging systems: An examination of tagging practices. 2006. Available from *http://dlist.sir.arizona.edu/1704/01/KippCampbell ASIST.pdf*.

[14] X. Li, L. Guo, and Y. E. Zhao. Tag-based social interest discovery. In *WWW*, 2008.

[15] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in XML. In *ICDE*, 2005.

[16] S. Michel, P. Triantafilou, and G. Weikum. Klee: a framework for distributed top-k query algorithms. In *VLDB*, 2005.

[17] A. Mislove, K. Gummadi, and P. Druschel. Exploiting social networks for internet search. In *HotNets*, 2006.

[18] J. Stoyanovich, S. Amer-Yahia, C. Marlow, and C. Yu. Leveraging tagging to model user interests in del.icio.us. In *AAAI SIP*, 2008.

[19] D. Watts and S. H. Strogatz. Collective Dynamics of "small-world" Networks. *Nature*, 393:440–442, 1998.

[20] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.

[21] D. Zhou, J. Bian, S. Zheng, H. Zha, and C. Giles. Exploring social annotations for information retrieval. In *WWW*, 2008.