Adding Structure to Top-K: From Items to Expansions

Xueyao Liang
Dept. of Computer Science,
Univ. of British Columbia
liangxue@cs.ubc.ca

Min Xie
Dept. of Computer Science,
Univ. of British Columbia
minxie@cs.ubc.ca

Laks V.S. Lakshmanan
Dept. of Computer Science,
Univ. of British Columbia
laks@cs.ubc.ca

ABSTRACT

Keyword based search interfaces are extremely popular as a means for efficiently discovering items of interest from a huge collection, as evidenced by the success of search engines like Google and Bing. However, most of the current search services still return results as a flat ranked list of items. Considering the huge number of items which can match a query, this list based interface can be very difficult for the user to explore and find important items relevant to their search needs. In this work, we consider a search scenario in which each item is annotated with a set of keywords. E.g., in Web 2.0 enabled systems such as flickr and del.icio.us, it is common for users to tag items with keywords. Based on this annotation information, we can automatically group query result items into different expansions of the query corresponding to subsets of keywords. We formulate and motivate this problem within a top-k query processing framework, but as that of finding the top-k most important expansions. Then we study additional desirable properties for the set of expansions returned, and formulate the problem as an optimization problem of finding the best k expansions satisfying all the desirable properties. We propose several efficient algorithms for this problem. Our problem is similar in spirit to recent works on automatic facets generation, but has the important difference and advantage that we don't need to assume the existence of pre-defined categorical hierarchy which is critical for these works. Through extensive experiments on both real and synthetic datasets, we show our proposed algorithms are both effective and efficient.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software - Information Networks

General Terms

Algorithms

Keywords

Top-*k* Query Processing, Top-*k* Query Expansions, Semantic Redundancy, Efficiency, Keywords and Tags

1. INTRODUCTION

Keyword based search interfaces are extremely popular as a means for efficiently discovering items of interest from a huge collection, as evidenced by the success of search engines like Google [5], Bing [2] and Yahoo! [7]. However, most of the current search services still return results as a flat ranked list of items. As has been found by a recent study [8], this list-based interface can make it very difficult for the user to explore and find important items relevant to their search needs. We claim that *automatically grouping search results into semantically independent "topics" can significantly enhance the usability of the results.* Below, we discuss a few motivating examples to illustrate this point.

Consider an academic search engine like Google Scholar [1]. Though it works perfectly for those queries which target a specific paper, when handling a general purpose query like "find all papers which are relevant to the topic of database histogram", the search engine returns a huge list of papers ranked by their relevance to the query. This ranked list is difficult for user to work from for exploring papers related to the query and efficiently finding the papers they want. It is clear that the user may benefit significantly if the search service can automatically group all the papers into semantically independent "topics". As a second example, consider search on social annotation websites like Del.icio.us [3] and Flickr [4]. These websites have rich user generated metadata for each item, however current search engines on these websites only utilize them to generate a ranked list of items for a query based on keyword relevance. In case of Del.icio.us, search results are presented using a faceted interface, but this is based on expanding the user's keyword query with one of a fixed set of tags. E.g., searching on the tag "programming" returns more than 1.3 million hits on Del.icio.us, however, only three single word tags "ajax", "software" and "javascript" are suggested to expand the query, while many other useful expansions such as "c++ programming tutorial" and "database programming language" cannot be found on the interface. As a third example, consider a community question answering forum like Quora [6]. A user may wish to search the Q&A in Quora using keywords and it is often helpful for the system to present the search results in an automatically grouped form, where different groups somehow correspond to different "subtopics" of the "topic" that the user may have questions about.

Indeed, to help users explore the returned search results, current search engines like Google, Bing and Yahoo! often add to the search result interface a set of facets, like publishing time, size of document, price etc. Faceted interfaces can greatly facilitate user navigation through the results. However, these facets are often predefined, and may not capture the attributes of the item which are the most important. E.g., for Google Scholar, current facets for the search results contain only types of publication and publication date, whereas the users may want to explore "topics" of papers among the search results. Similar comments apply to the other two example search scenarios above.

Our problem is similar in spirit to recent works on automatic facets generation [19], but has the important difference and advantage that we don't need to assume the existence of pre-defined categorical hierarchy which is critical for these works. Indeed, in the applications we consider such as above, we cannot assume any prior taxonomy.

Motivated by these drawbacks of current search result interfaces, we consider a search scenario in which each item is annotated with a set of keywords. These can be keywords associated with papers or tags assigned to items by users of social annotation systems or keywords occurring in question and answers in Q&A systems. Based on this annotation information, we want to automatically group query result items into different expansions of the query corresponding to subsets of keywords. Items may have a number of attributes, either explicitly stored or computed, which can signify their importance or utility to the user. E.g., papers have citation score, pagerank of their authors, etc. In social annotation systems, popularity of items and pagerank of the annotator can be important attributes. Finally, in a system like Quora, we have attributes like importance scores of questioner or answerer, number of people interested in a question, etc. Intuitively, the expansions that we wish to return to a user should be driven by how important the items are that match an expansion and how many such important items match it. This raises the problem of what expansions of a query should we show the user? We formulate and motivate this problem within a top-k query processing framework, but as that of finding the top-k most important expansions, where the importance or utility of an expansion is driven by the utility of the items matching it.

We make the following contributions:

- We introduce the problem of finding top-k high quality expansions (Section 2).
- We first propose a naïve algorithm for this problem and then discuss its limitations and propose a significant improvement leveraging the lattice structure of expansions (Section 3).
- We consider some semantic issues with directly returning the top-k high quality expansions, and propose several different algorithms for improving the quality of the returned results (Section 4).
- Through extensive experiments on both real and synthetic datasets, we demonstrate that our proposed algorithms have excellent performance and very good quality (Section 5).

Related work is discussed in Section 6. We summarize the work in Section 7 and discuss open research problems.

2. PROBLEM DEFINITION

Consider a set of items $S = \{t_1, ..., t_n\}$, each item $t_i \in S$ being associated with a set of m attributes $\{a_1, ..., a_m\}$. We denote the value of t_i on attribute a_j as $t_i.a_j$, and assume without loss of generality that all values on attribute a_j are normalized to [0, 1], $j \in 1...m$. Attributes of an item t_i correspond to t_i's "utility" to a user, and in this work, without loss of generality, we assume large attribute values are preferred, so larger the value larger the utility. The overall *utility* of an item t_i, denoted $u(t_i)$, is captured by a weighted sum of its values on all attributes, i.e., $u(t_i) = \sum_{j \in 1...m} w_j \times t_i.a_j$, where w_j is a positive weight associated with attribute a_j. These weights may be chosen by a user or the system may learn the "best" weights from user behavior using models like linear regression [10].

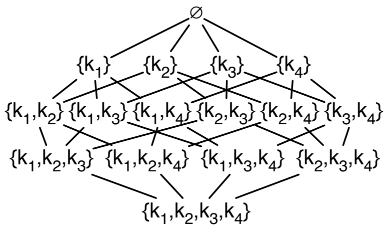

∅
{k1} {k2} {k3} {k4}
{k1,k2} {k1,k3} {k1,k4} {k2,k3} {k2,k4} {k3,k4}
{k1,k2,k3} {k1,k2,k4} {k1,k3,k4} {k2,k3,k4}
{k1,k2,k3,k4}

Figure 1: Lattice structure of expansions.

We assume each item $t_i \in S$ is also associated with a set of l keywords or tags $\{k_1, ..., k_l\}$ which summarize the contents of t_i. We denote the set of keywords associated with an item t_i as $\mathrm{Kw}(t_i)$. A query Q in our system is a set of keywords, $Q = \{k_1, ..., k_p\}$. An item t *matches* query Q iff $Q \subseteq \mathrm{Kw}(t)$. The answer to Q, denoted S_Q, is then the set of matching items in S, i.e., $S_Q = \{t \mid t \in S, Q \subseteq \mathrm{Kw}(t)\}$.

As mentioned in the introduction, the number of items matching a query Q can often be huge, and merely returning the top-k matching items may not help the user find the items they are most interested in. So we want to group items into different expansions of Q and return high quality expansions. Let K denote the set of all keywords in the system. We call a subset of keywords $e \subseteq K - Q$ an *expansion*[1] of Q. The size of an expansion e, $|e|$, is the number of keywords in e. Let the set of all possible expansions for Q be E_Q, then given an expansion $e \in E_Q$, we say an item $t \in S$ *matches* e if $Q \cup e \subseteq \mathrm{Kw}(t)$. Let S_e denote the set of matching items for e, then we have $S_e = \{t \mid t \in S, Q \cup e \subseteq \mathrm{Kw}(t)\}$.

Given a query Q, all possible expansions of Q can be organized as a lattice based on the *subset-of* relationship, E.g., Figure 1 shows the lattice structure of all possible expansions for $K - Q = \{k_1, ..., k_4\}$. We will shortly define the quality of an expansion, and we will show in Section 3.2 that this lattice structure can be used to improve the efficiency of our algorithm for determining high quality expansions.

Intuitively, the importance of an expansion e can be captured by aggregating the utilities of items which match e. Let g be a monotone aggregation function which calculates the overall utility or quality of expansion e, i.e., define $u(e) := g(\{u(t) \mid t \in S_e\})$. Then we can define our top-k expansion problem as follows.

Definition 1 (Top-k Expansions). *Given a set S of items and a keyword query Q, find the top-k expansion set* $E_k = \{e_1, ..., e_k\}$ *s.t.* $\forall e \in E_k$ *and* $\forall e' \in E_Q - E_k$, $u(e) \geq u(e')$.

The intuition is that in response to a user query, we want to return the best quality expansions where the quality of an expansion is monotonically determined by the utility of the items matching it.

The set of annotations used in this paper is summarized in Table 1. Below, we give an example of an aggregation function with certain desirable properties.

2.1 Determining Importance of An Expansion

To determine the utility $u(e)$ of expansion e, a natural idea is to use a monotone function to aggregate utility values of all items which can match e. However, this approach means for every expansion, we may need to retrieve all items that are relevant to this expansion which can be prohibitive considering the huge size of matching items. Besides, low quality items matching e intuitively should not determine its importance. So we will only consider top-

[1]Note, actually the real expansion is $Q \cup e$ but for technical convenience, we deal with the part of the expansion outside Q.

Symbol	Description
t	an item in the system
S	a set of items
$t.a$	attribute a's value of an item t
w	the weight associated with an attribute
$u(t)$	the utility value of an item t
k	a keyword
K	the set of all possible keywords
$\mathcal{L}_K$	the lattice of all possible keywords K
$\mathcal{L}$	the partially materialized lattice
Kw(t)	the set of keywords associated with t
Q	a query which is composed of a set of keywords
S_Q	the set of matching items for query Q
e	an expansion
E_Q	the set of all possible expansions for query Q
S_e	the set of matching items for expansion e
$u(e)$	the utility value of an expansion e
V_{S_e}	the multiset of utility values of all items matching e
$\mathcal{L}_K(e)$	the sub-lattice induced by e
E_t	all possible expansions of t
e_t	the largest size expansion in E_t

Table 1: Annotations

N matching items for determining importance of an expansion e, where N is a parameter that is tuned for each application.

Let $V_{S_e} = \{u(t) \mid t \in S_e\}$ be the multiset of utility values of all items matching e. We will apply a function top_N to V_{S_e} which retrieves the top-N highest utility values from V_{S_e}. Then to determine the importance of e, we will sum up all values in $top_N(V_{S_e})$, so $g(S_e) := \sum_{v \in top_N(V_{S_e})} v$. It is clear that $g(S_e)$ satisfies the desired properties for determining the importance of an expansion, as $g(S_e)$ will be large when there are high quality items matching e and also when the number of these high quality items is large.

It can be easily shown that the aggregation function g defined is *subset-monotone*, which means for two expansions e_1 and e_2, if $S_{e_1} \subseteq S_{e_2}$, then $g(e_1) \leq g(e_2)$. And it is worth noting that the algorithms proposed in this work can be easily adapted to other subset-monotone score functions.

On top of the basic top-k expansion problem, we will want to impose some additional desirable properties for the k expansions returned by our algorithms. In Section 4, we study these properties and propose additional algorithms which can return expansions satisfying these properties.

3. BASIC ALGORITHM

In this section, we present two algorithms for finding top-k expansions. In order to facilitate both algorithms, we assume m inverted lists are materialized, one each for the m attributes, where for each inverted list, items are sorted in the non-increasing order of their value on the corresponding attribute.

3.1 Naïve Algorithm

Inspired by the NRA algorithm proposed in [13], a naïve way of generating top-k expansions can be described as follows: 1. access items in the non-increasing order of their attribute value; 2. for each matching item accessed, enumerate all possible expansions and update their lower bound and upper bound utility value; 3. stop the iterative process once top-k expansions have been identified. The pseudo-code for the above process is given in Algorithm 1.

In Algorithm 1, all attribute lists are accessed in a round-robin fashion (line 4), and for each matching item t obtained from list I_a, we will enumerate the set E_t of all possible expansions (line 5–7).

Consider an expansion $e \in E_t$, let SS_e be the current set of accessed items which can match e. Then for each $t \in SS_e$, because all attribute values are normalized to $[0, 1]$ and items are accessed in the non-increasing order of their attribute values, similar to the NRA algorithm [13], we can determine the lower bound on t's utility $\underline{u}(t)$ by summing up all current accessed attribute values of t, and we can determine its upper bound utility $\bar{u}(t)$ by summing up all accessed attribute values of t along with the last accessed values of other attributes in the inverted lists.

Then because of the subset-monotonicity of the score function g, the lower bound utility $\underline{u}(e)$ for expansion $e \in E_t$ can be calculated as a sum of all values in $top_{min(N,|SS_e|)}(\{\underline{u}(t) \mid t \in SS_e\}))$. And the upper bound utility $\bar{u}(e)$ of an expansion e can be estimated by considering both SS_e and the maximum utility value which can be achieved by any unseen items. Let the last accessed values on each of the m attribute lists be $\bar{v}_1, ..., \bar{v}_m$ respectively, then because items are accessed in the non-increasing order of their attribute value, an "imaginary" item t' with m attribute values $\bar{v}_1, ..., \bar{v}_m$ must have the maximum value that can be achieved by any unseen items. So the upper bound utility of e can be estimated as the sum of all values in $top_N(\{\bar{u}(t) \mid t \in SS_e\} \cup \{u(t'_1), ..., u(t'_N)\}))$. where $t'_1, ..., t'_N$ are N imaginary items which have the same utility as t'.

After the lower and upper bounds for all expansions in E_t have been updated (line 8–10), we can estimate the upper bound utility for all possible unseen items by summing up the values of N maximum possible imaginary items (updateUpperBound($\emptyset$) in line 12).

Henceforth, by the value of an expansion, we mean its utility value. Let EQ be the expansion query which contains the set of expansions of Q that have been materialized by the algorithm at a given point. We can rank all expansions in EQ by their lower bound values. Let EQ_k be the top-k expansions in EQ and UB' be the maximum upper bound value of an expansion in $EQ - EQ_k$, then the maximum of UB' and the upper bound value for all unseen items determines the overall upper bound value UB (line 11–12). If the lower bound value for the k^{th} expansion in EQ_k is already larger than UB, the algorithm can be safely terminated, exactly in the spirit of NRA.

Algorithm 1: TopExp-Naive(Q, I, g, k)

```
1  $EQ \leftarrow$ expansion queue;
2  $UB \leftarrow$ upper bound threshold;
3  while $|EQ| < k$ OR $u(EQ.k^{th}$ expansion$) < UB$ do
4      $I_a \leftarrow$ getNextListRR();
5      $t \leftarrow I_a$.getNextItem();
6      if $Q \not\subseteq$ Kw($t$) then continue;
7      $E_t \leftarrow$ enumerateExpansion(Kw($t$) $-$ $Q$);
8      foreach $e \in E_t$ do
9          if $e \notin EQ$ then $EQ$.push($e$);
10         $e$.updateLower&UpperBound($t$, $g$);
11     $UB' \leftarrow$ maximum upper bound value of an expansion in
       $EQ - EQ_k$;
12     $UB \leftarrow$ MAX($UB'$, updateUpperBound($\emptyset$));
```

The correctness of Algorithm 1 easily follows from the fact that both lower bound and upper bound of an expansion are correct. And to assess the performance of Algorithm 1, we borrow the notion of *instance optimality* as proposed by Fagin et al. in [13].

Definition 2. **Instance Optimality:** *Let $\mathcal{A}$ be a class of algorithms that make no random accesses to the m inverted lists, and let $\mathcal{I}$ be a class of problem instances. Given a non-negative cost measure cost(A, I) of running algorithm $A \in \mathcal{A}$ over $I \in \mathcal{I}$, an algorithm $A \in \mathcal{A}$ is* instance optimal *over $\mathcal{A}$ and $\mathcal{I}$ if for every*

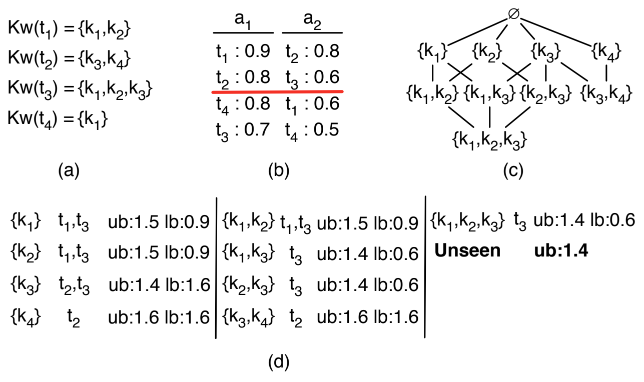
$A' \in \mathcal{A}$ and every $I \in \mathcal{I}$ we have $cost(A, I) \leq c \cdot cost(A', I) + c'$, for constants c and c'. Constant c is called the optimality ratio.

Let the cost of an algorithm for the top-k expansion problem be determined by the number of items accessed, we first show the following result.

Lemma 1. *Given any instance I of the top-k expansion problem and any algorithm A with the same access constraints as TopExp-Naive, assume A accesses x items, then TopExp-Naive will access at most $m \times x$ items.*

Proof. (Sketch) Assume another algorithm $B \in \mathcal{A}$ stops in list I_a after accessing x items, then it must be true that at the time when B stops, the current utility of the k^{th} maximum utility expansion e_k will have its utility larger than or equal to the upper bound utility of all generated non-result expansions and all possible unseen expansions. This is because otherwise, we can come up with an configuration of the remaining unseen items such that we can have an expansion of which the utility is larger than e_k. Then it is clear that our algorithm TopExp-Naive will also stop at the same position in list I_a. By considering the fact there are m lists in total, we can infer that the total number of items accessed by TopExp-Naive is at most $m \times x$, for the scenario where B accesses x items in list I_a, zero item in other lists, and TopExp-Naive accesses x items in each list. $\square$

Theorem 1. *Let $\mathcal{I}$ be the class of all top-k expansion problem instances, and $\mathcal{A}$ be the class of all possible algorithms that find the top-k expansions, that are constrained to access items sequentially in non-increasing order of their attribute values, then TopExp-Naive is instance optimal over $\mathcal{A}$ and $\mathcal{I}$ with an optimality ratio of m.*



a1	a2
t1 : 0.9	t2 : 0.8
t2 : 0.8	t3 : 0.6
t4 : 0.8	t1 : 0.6
t3 : 0.7	t4 : 0.5

{k1}	t1,t3	ub:1.5 lb:0.9	{k1,k2}	t1,t3	ub:1.5 lb:0.9	{k1,k2,k3}	t3	ub:1.4 lb:0.6
{k2}	t1,t3	ub:1.5 lb:0.9	{k1,k3}	t3	ub:1.4 lb:0.6	**Unseen**		**ub:1.4**
{k3}	t2,t3	ub:1.4 lb:1.6	{k2,k3}	t3	ub:1.4 lb:0.6			
{k4}	t2	ub:1.6 lb:1.6	{k3,k4}	t2	ub:1.6 lb:1.6			

Figure 2: Example for TopExp-Naive.

Example 1. We show an example of algorithm TopExp-Naive in Figure 2. In this example, for simplicity of presentation, we assume all 4 items $t_1, ..., t_4$ can match the query Q, so Q will be ignored from the keyword list of all items. Furthermore, we assume $N = 1$ which means we are using the best item to determine the importance of each expansion, and $k = 1$ which means we are looking for top-1 expansion. Keywords associated with each item are shown in Figure 2 (a), and each item in the example has two attributes a_1 and a_2. The inverted lists for these two attributes are shown in Figure 2 (b).

As described in TopExp-Naive, the algorithm will enumerate all possible expansions for each item accessed, e.g., after t_1 is accessed, $\{k_1\}$, $\{k_2\}$ and $\{k_1, k_2\}$ will be generated, and their utility bound values will be updated using the attribute value of t_1. For this case, because only $t_1.a_1$ is known, we know $\bar{u}(t_1) = 1.9$ and $\underline{u}(t_1) = 0.9$, these will be the upper and lower bound utility value for all three expansions. After we have accessed the first two items of both lists, the expansions generated are shown in Figure 2 (c), while the items contained in each generated expansion and the utility bound values for each generated expansion are shown in Figure 2 (d). It is clear that at this moment, the upper bound utility for all generated expansions is 1.6, the upper bound utility for all unseen expansions is 1.4 (sum of the last accessed utility value from each list) and the lower bound utility for expansions $\{k_3\}$, $\{k_4\}$ and $\{k_3, k_4\}$ are all 1.6, so we can stop the algorithm now and return any one of these three expansions. $\square$

3.2 Improved Algorithm

One serious drawback of the naïve algorithm is that every time an item t containing keywords Kw(t) is accessed, all $2^{|\text{Kw}(t)-Q|}$ possible expansions for this item are explicitly enumerated and their bounds are maintained. In this section, we propose an efficient algorithm which can leverage the lattice structure of expansions to avoid enumerating and maintaining unnecessary expansions. We also address the challenge of determining the bounds of unseen (unmaterialized) expansions, which is necessary for early termination of the top-k algorithm.

3.2.1 Avoiding Unnecessary Expansions

Given a query Q and a newly accessed item t, let e_t = Kw(t) − Q be the largest size expansion in E_t, then the naïve algorithm will enumerate all possible expansions of t by considering all non-empty subsets of e_t. However, this may not be necessary. E.g., let $K_{<t}$ be the set of keywords which have been seen before t; if $\forall k \in K_{<t}$, $k \notin e_t$, we just need to maintain one single expansion e_t, as all other expansions generated from e_t will have the same current matching itemset as e_t and thus the same lower and upper bounds as e_t. This indicates that there are opportunities to avoid the expansion enumeration process for each newly accessed item.

Let $\mathcal{L}_K$ be the lattice of all possible keywords K. For an expansion $e \in \mathcal{L}_K$, we let $\mathcal{L}_K(e)$ denote the sub-lattice *induced* by e, i.e., $\mathcal{L}_K(e) = \{e' \mid e' \subseteq e\}$. For a newly accessed item t, the naïve algorithm will enumerate all expansions $e \in \mathcal{L}_K(e_t)$, however, as discussed above that this isn't always necessary.

The idea of the new algorithm can be described as follows. Let $\mathcal{L}$ denote a partially materialized lattice which contains the set of expansions generated so far before the current item t. If $\exists e \in \mathcal{L}$ s.t. $e = e_t$, then it is clear we just need to update the lower bound and upper bound utilities of all existing expansions in $\mathcal{L}$ which are subsets of e_t. Otherwise, all expansions in $\mathcal{L}$ will correspond to different sets of items compared with e_t, so we need to first generate the expansion e_t and update its lower bound and upper bound utility. Then we consider the following two cases: 1. if $\forall e \in \mathcal{L}$, $e \cap e_t = \emptyset$, then all $e' \in \mathcal{L}_K(e_t)$ correspond to the same set of items, their lower bound and upper bound utilities are the same, and we just need to maintain one expansion e_t which can concisely represent all expansions in $\mathcal{L}_K(e_t)$; 2. On the other hand, if there exists an expansion $e \in \mathcal{L}$ s.t. $e \cap e_t \neq \emptyset$, then *for each such expansion e*, we need to further consider the following three sub-cases:

1. if $e \subseteq e_t$, we don't need to generate additional expansions, but we need to update the lower bound and upper bound utility of e since item t also contributes to every sub-expansion e of e_t.
2. if $e_t \subseteq e$, similar to case 1, we don't need to generate additional expansions; and since t cannot contribute to e, we don't need to update utility bounds of e.

Algorithm 2: TopExp-Lazy(Q, I, g, k)

```
1  $UB \leftarrow$ upper bound threshold;
2  $\mathcal{L} \leftarrow$ partial materialized lattice structure;
3  while $|EQ| < k$ OR $u(EQ.k^{th}$ expansion$) < UB$ do
4      $I_a \leftarrow$ getNextListRR();
5      $t \leftarrow I_a$.getNextItem();
6      if $Q \not\subseteq \text{Kw}(t)$ then continue;
7      $e_t \leftarrow \text{Kw}(t) - Q$;
8      if $e_t \in \mathcal{L}$ then
9          foreach $e \in \{e \mid e \in \mathcal{L} \land e \subseteq e_t\}$ do
10             $e$.updateLower&UpperBound($t$);
11     else
12         $TQ \leftarrow$ temporary update expansion queue;
13         $TQ$.push($e_t$);
14         while $\neg$ $TQ$.empty() do
15             $e = TQ$.pop();
16             $E_l \leftarrow \{e' \mid e' \in \mathcal{L} \land (\nexists e'' \in \mathcal{L} : e' \subset e'')\}$;
17             foreach $e_l \in E_l \land e_l \cap e \neq \emptyset$ do
18                 $E \leftarrow \{e' \mid e' \in \mathcal{L} \land e' \subseteq e_l\}$;
19                 if $\exists e' \in E$ s.t. $e' \not\subseteq e \land e \not\subseteq e'$ then
20                     Find all $e' \in E$ s.t. $e' \not\subseteq e \land e \not\subseteq e'$ and
                       $(\nexists e'' \in E : e' \subset e'' \land e'' \not\subseteq e \land e \not\subseteq e'')$;
21                     $TQ$.push($e \cap e'$);
22             if $e \notin \mathcal{L}$ then $\mathcal{L}$.add($e$);
23         foreach $e \in \mathcal{L} \land e \subseteq e_t$ do
24             $e$.updateLower&UpperBound($t$);
25     $UB \leftarrow$ upper bound value of the $(k+1)^{th}$ expansion in $\mathcal{L}$;
26     $UB \leftarrow$ MAX($UB$, getUpperBound($\emptyset$));
```

3. if $e \not\subseteq e_t$ or $e_t \not\subseteq e$, let $e' = e \cap e_t$, then it is clear that $S_{e'} \neq S_e$ and $S_{e'} \neq S_{e_t}$, which means we need to generate a new expansion e' and update its utility bounds accordingly.

3.2.2 Bounds for Unseen Expansions

Since we don't explicitly maintain all possible expansions for each item accessed, this will create a challenge for determining lower bound and upper bound utilities for all possible expansions. For all expansions which are maintained in $\mathcal{L}$, the lower bound and upper bound are determined as discussed in Section 3.1. For each remaining expansion e not materialized in $\mathcal{L}$, depending on the position of e in the lattice $\mathcal{L}_{\mathcal{K}}$, we need to consider the following two cases:

- $\nexists e' \in \mathcal{L}$ s.t. $e \subset e'$. This means we haven't accessed any item which corresponds to this expansion. The utility upper bound of e in this case is the same as the maximum possible utility of an unseen expansion, as discussed in Section 3.1.
- $\exists e' \in \mathcal{L}$ s.t. $e \subset e'$. This means we have already accessed some items which correspond to this expansion. Then we must be able to find a smallest such expansion $\hat{e} \in \mathcal{L}$ s.t. $e \subset \hat{e}$, and $\forall e' \in \mathcal{L}$, if $e \subset e'$, then $\hat{e} \subseteq e'$. To see this, suppose $e', e'' \in \mathcal{L}$ are distinct expansions such that $e \subset e'$ and $e \subset e''$. From Section 3.2.1, it follows that for $e''' = e' \cap e''$, we have $S_{e'''} \neq S_{e'}$ and $S_{e'''} \neq S_{e''}$, so e''' should be generated as a new expansion before e, and this expansion has the property that it is a subset of both e' and e'', and it is a superset of e. It follows that the smallest superset expansion $\hat{e}$ must exist in $\mathcal{L}$. So after $\hat{e}$ is found, we know e and $\hat{e}$ currently correspond to the same set of items, then e's utility bound will be the same as $\hat{e}$. This means that we don't need to explicitly consider this expansion when using the utility bounds to determine whether the algorithm can stop.

Example 2. Consider the lattice in Figure 1 and assume we have only materialized two expansions $e_1 = \{k_1\}$ and $e_2 = \{k_1, k_2, k_3\}$. Then for an expansion $e_3 = \{k_2, k_4\}$, because there is no such materialized expansion $e' \in \mathcal{L}$ s.t. $e_3 \subset e'$, then we know we haven't accessed any item which corresponds to this expansion. So we only need to consider its upper bound utility, which is the maximum possible utility for all possible expansions. And for another unmaterialized expansion $e_4 = \{k_1, k_2\}$, we can find out that $e_4 \subset e_2$, so e_4 and e_2 correspond to the same set of items, and e_4's utility bounds are the same as those of e_2. □

So the general idea of our *lazy expansion generation* based algorithm is that we only need to maintain expansions which correspond to a *unique* set of items. For a set of expansions which are matched by the same set of items, we can simply represent them using the largest expansion in the set.

3.2.3 Lazy Expansion Algorithm

The pseudo-code for the lazy expansion generation based algorithm is given in Algorithm 2. Similar to TopExp-Naive, TopExp-Lazy iteratively retrieves items from the attribute lists (line 4–6). However, unlike TopExp-Naive, *we maintain only necessary expansions in the partially materialized lattice $\mathcal{L}$*. For a newly accessed item t, if e_t has already been generated by a previous item, we simply update the itemset and lower/upper utility bounds of the corresponding expansions which contain this item (line 8–10). Otherwise, as discussed above, we may need to generate some additional expansions which correspond to a unique matching itemset (line 11–24). The procedure works as follows. First, we identify from $\mathcal{L}$ all *leaf expansions E_l* which are maximal expansions, i.e., they are not included in other expansions in $\mathcal{L}$ (line 16). Then for each leaf expansion $e_l \in E_l$, if e_t doesn't overlap with e_l, we can ignore all expansions which are subsets of e_l as they can't overlap with e_t either (line 17). If e_l overlaps with e_t, then we search the set E of expansions in $\mathcal{L}$ which are subsets of e_l: if there is an expansion $e' \in E$ s.t. $e' \not\subseteq e_t$ and $e_t \not\subseteq e'$, then we need to find all of the largest such expansions e' in E, and as described in Section 3.2.1, for each of them, a new expansion $e' \cap e_t$ needs to be inserted to $\mathcal{L}$ as it corresponds to a unique set of matching items, so we will recursively insert this new expansion into $\mathcal{L}$ (line 19–21).

The procedure for updating lower bound and upper bound utilities for each expansion is very similar to TopExp-Naive. However, because in TopExp-Lazy each expansion may represent more than one expansion, in order to determine which expansions in the expansion buffer $\mathcal{L}$ are current top-k expansions, we need to calculate for each expansion $e \in \mathcal{L}$ the exact number of ungenerated expansions which have the same utility bounds as e. The pseudo-code for this procedure are listed in Algorithm 3 and Algorithm 4.

The idea of Algorithm 3 is that we first find from $\mathcal{L}$ the expansion set ST_e of which the expansions are subset of e (line 1), we prune away those expansions in ST_e of which a superset is also present in ST_e (line 2–3), then for the pruned expansion set ST_e, we can use the classical *inclusion-exclusion principle* to count the total number of expansions covered by ST_e (line 4). Algorithm 4 is a simple implementation of the counting procedure.

Example 3. Figure 3 shows an example that illustrates how Algorithm TopExp-Lazy works. The configuration of this example, including query, item attributes, item attribute values, keywords of

Algorithm 3: updateCount($\mathcal{L}$, e)

```
1 $ST_e \leftarrow \{e' \mid e' \in L \wedge e' \subset e\}$;
2 $ST'_e \leftarrow \{e' \mid e' \in ST_e \wedge \exists e'' \in E \rightarrow e' \subset e''\}$;
3 $ST_e = ST_e - ST'_e$;
4 $count \leftarrow$ countGeneratedExpansions($ST_e$);
5 $e$.count $\leftarrow 2^{|e|} - count - 1$;
```

Algorithm 4: countGeneratedExpansions(ST_e)

```
1  $count \leftarrow 0$;
2  for $outeridx$ from 2 to $|ST_e|$ do
3      $ST'_e = \emptyset$;
4      for $inneridx$ from 1 to $outeridx$ - 1 do
5          $ST'_e$.insert($ST_e[outeridx] \cap ST_e[inneridx]$);
6      $ST''_e \leftarrow \{e' \mid e' \in ST'_e \wedge \exists e'' \in E \rightarrow e' \subset e''\}$;
7      $ST'_e = ST'_e - ST''_e$;
8      if $ST'_e$.hasOverlap() then
9          $count$ += countGeneratedExpansions($ST'_e$);
10     else
11         foreach $e' \in ST'_e$ do
12             $count$ += $2^{|e'|}$-1;
```

each item and parameters k, N, is the same as Example 1. However, because we are using the lazy expansion generation based algorithm, we don't need to enumerate all possible expansions for each item accessed. E.g., when the first item t_1 is accessed, we only need to generate expansion $\{k_1, k_2\}$ and don't need to generate expansions $\{k_1\}$ and $\{k_2\}$, as they correspond to the same current set of matching items as $\{k_1, k_2\}$. The utility bound values for $\{k_1, k_2\}$ will be the same as in the TopExp-Naive algorithm, and again we don't need to maintain these utility bound values for $\{k_1\}$ and $\{k_2\}$ since they are the same as for $\{k_1, k_2\}$. After accessing two items from each lists, the expansions materialized for TopExp-Lazy are shown as bolded expansions in Figure 3 (c). Compared with TopExp-Naive, it's worth noting that 5 expansions don't need to be maintained. At this point, similarly to TopExp-Naive, the algorithm can also stop as the top expansion $\{t_3, t_4\}$'s lower bound utility is already larger than or equal to the maximum upper bound utility for all expansions.

Note that for the lazy expansion generation based algorithm, for each expansion which needs to materialized, we need to use Algorithm 3 to count how many expansions correspond to the same set of items. E.g., after accessing t_3 in the inverted list of a_2, we need to consider how many expansions are "covered by" the current expansion $\{k_1, k_2, k_3\}$. Algorithm 3 will first find all materialized expansions which are subsets of $\{k_1, k_2, k_3\}$, for this case, $\{k_1, k_2\}$ and $\{k_3\}$. Then as in line 4 of Algorithm 3, Algorithm 4 will be called to enumerate the number of non-empty expansions covered by these two expansions. Because there is no overlap between $\{k_1, k_2\}$ and $\{k_3\}$, in line 10–12 of Algorithm 4, we can simply sum up the number of non-empty expansions covered by these two expansions, which is 4. Then this number will be used to determine the number of non-empty expansions covered by $\{k_1, k_2, k_3\}$ in line 5 of Algorithm 3, which is 3 ($\{k_1, k_2, k_3\}$, $\{k_1, k_3\}$ and $\{k_2, k_3\}$).

Figure 3 (d) shows all the expansions that should be considered and whether they are "covered by" some other expansions in the lattice. Compared with Figure 2 (d), it is clear that the lazy expansion generation based algorithm will maintain just one expansion for each set of expansions which correspond to the same set of items. □

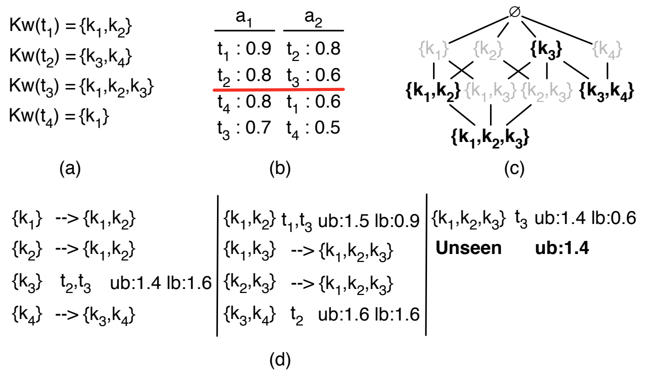


Figure 3: Example for TopExp-Lazy.

4. SEMANTIC OPTIMIZATION

Though algorithms described in Section 3 can correctly find expansions which have the k highest utility, there are two kinds of issues with these algorithms. First, the basic algorithm will favor small expansions (i.e., fewer keywords) as these expansions have more matching items than larger expansions. Second, in the returned top-k expansions, it may happen that two expansions have the subset-of relationship, which is not ideal. Indeed, we would like the resulting expansions to have little overlap with each other.

In this section, we propose two solutions to remedy the above drawbacks. In Section 4.1, we will study weighting schemes which can penalize expansions that are either too small or too large. Then in Section 4.2, we propose to find the k most interesting expansions which don't have overlap with each other.

4.1 Weighting Expansions

It is clear that expansions which have small size (e.g., "XML") correspond to "general topics" which are related to the query, whereas expansions which have large size (e.g., "XML, schema, conformance, automata") correspond to "specific topics" which are related to the query. To help users quickly locate interesting information from the returned results, intuitively neither too general topics nor too specific topics should be returned early, so we want to favor those expansions which are neither too large nor too small.

We assume there is a function $f_w : \mathbb{N} \rightarrow \mathbb{R}$ (where $\mathbb{N}/\mathbb{R}$ denote the sets of natural/real numbers) which can return the weight $f_w(p)$ for an expansion of size p. The intuition is that f_w penalizes too small and too large expansions for we expect them to be intuitively too general or too specific. Then we can use this function to weight the utility of all expansions under consideration. In this work, we consider the Gaussian function $f_w(p) = e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. The mean μ of f_w is set as the most ideal size of an expansion and the variance σ can be adjusted by the system for different problem instances.

So how are the top-k algorithms impacted? For TopExp-Naive, the weighting function can be simply applied to the utility bounds of each expansion enumerated, and other parts of the algorithm won't be affected. However, for TopExp-Lazy, for each materialized expansion $e \in \mathcal{L}$, the unmaterialized expansions which have the same set of matching items as e have the same utility bounds as e when no weighting is applied. But once weighting is applied, these expansions may have different utility bounds depending on the size. E.g., consider that there is only one materialized expansion $e = \{k_1, k_2\}$ in $\mathcal{L}$ and let the lower and upper utility bounds of e be $\underline{u}(e) \times f_w(2)$ and $\bar{u}(e) \times f_w(2)$ respectively. Then for expansions $\{k_1\}$ and $\{k_2\}$, though they correspond to the same set of matching items as e, their lower and upper utility bounds are $\underline{u}(e) \times f_w(1)$ and $\bar{u}(e) \times f_w(1)$ respectively.

So for the weighted lazy expansion generation based algorithm, for a set E of expansions which correspond to the same set of matching items, we may need to maintain multiple expansions where the number of expansions to be maintained depends on the size of the largest expansion in E. For an expansion e, Algorithm 5 and Algorithm 6, which are adapted from Algorithm 3 and Algorithm 4, can be used to count for each possible expansion size, the number of expansions which correspond to the same set of matching items as e. These counts can be utilized along with the weighting function to determine the corresponding lower and upper utility bounds.

Algorithm 5: updateCount(λ, e)

```
1 $ST_e \leftarrow \{e' \mid e' \in L \wedge e' \subset e\}$;
2 $ST'_e \leftarrow \{e' \mid e' \in ST_e \wedge \exists e'' \in E \rightarrow e' \subset e''\}$;
3 $ST_e = ST_e - ST'_e$;
4 $vec_c \leftarrow$ a new count vector of size $|e|$;
5 for $expsize$ from 1 to $|e|$ do
6     $vec_c[expsize] = \binom{|e|}{expsize}$;
7 $vec_c \leftarrow$ countGeneratedExpansions($ST_e, vec_c$);
8 $e$.count = $vec_c$;
```

Algorithm 6: calculateSameLBExpansionVec(ST_e, vec_c)

```
1  for outeridx from 2 to $|ST_e|$ do
2      $ST'_e = \emptyset$;
3      for inneridx from 1 to outeridx - 1 do
4          $ST'_e$.insert($ST_e[outeridx] \cap ST_e[inneridx]$);
5      $ST''_e \leftarrow \{e' \mid e' \in ST'_e \wedge \exists e'' \in E \rightarrow e' \subset e''\}$;
6      $ST'_e = ST'_e - ST''_e$;
7      if $ST'_e$.hasOverlap() then
8          countGeneratedExpansions($ST'_e$, $vec_c$);
9      else
10         for expsize from 1 to $|e|$ do
11             foreach $e' \in ST'_e$ do
12                 $vec_c[expsize] = vec_c[expsize]$ - $\binom{|e'|}{expsize}$;
```

4.2 Path Exclusion based Algorithm

To lessen the semantic overlap between different expansions returned to the user, intuitively we may not want to return two different expansions e_1 and e_2, such that either $e_1 \subset e_2$ or $e_2 \subset e_1$. Note that there can be many sets of expansions satisfying this pairwize comparability, and the set of highest utility expansions may not satisfy this property. So in order to guarantee the quality of the expansions returned, we want to maximize the sum of the utilities for the set of expansions returned under the constraint that the expansions returned should satisfy the pairwise comparability.

Definition 3. *(Maximum k Path-Exclusive Expansion) Given a set S of items and a keyword query Q, find the top k-expansion set $E_k = \{e_1, ..., e_k\}$ s.t. $\forall e_i, e_j \in E_k$, $i \neq j$, $e_i \not\subset e_j$, $e_j \not\subset e_i$, and $\sum_{e \in E_k} u(e)$ is maximized.*

Let $\mathcal{L} = \{e_1, ..., e_n\}$ be the set of all expansions materialized by the algorithm. Consider a weighted undirected graph $G = (Vtx, Edg)$, with nodes $Vtx = \mathcal{L}$ where each node e_i is associated with a weight $u(e_i)$, i.e., the utility of e_i. Whenever two expansions e_i, $e_j \in Vtx$ are such that either $e_1 \subset e_2$ or $e_2 \subset e_1$, Edg contains the edge (e_i, e_j). Then it is straightforward to show that the maximum k path-exclusive expansion problem is NP-hard by a direct reduction from the *maximum weighted independent set* problem [18].

A simple greedy algorithm for the maximum weighted independent set was proposed by [18]: repeatedly select a node in G with minimum weighted degree in each iteration and add it to the current solution; then delete this node and all of its neighbors from the graph; stop when all nodes are removed from G. It has been proven in [18] that this algorithm gives a $\max(\delta_w, 1)$-approximation, where $\delta_w = \max_{H \subseteq G} \min_{v \in V(H)} d_w(v, H)$, $d_w(v, H)$ is defined as $\frac{ww(N_G(v))}{ww(\{v\})}$, $ww(S)$ denotes the sum of weights of a set of nodes S, and $N_G(v)$ is the set of neighbors of v in G.

Furthermore, if we rank all generated expansions by their upper bound utility, because items are accessed in the non-increasing order of their attribute values, it is clear that the sum of the top-k expansions' upper bound utilities is an upper bound for the value of all possible k path-exclusive expansions.

So based on this information, we propose the following algorithm called Top-PEkExp, which can be used to calculate an approximate solution for the maximum k path-exclusive expansion problem. In Top-PEkExp, similar to the previous algorithms, we iteratively retrieve items from the attribute lists (line 3–5), then we use TopExp-Lazy to generate necessary expansions in $\mathcal{L}$ (line 6). The set of expansions in $\mathcal{L}$ are sent to the greedy algorithm for the maximum weighted independent set problem (line 7), and if the result is already larger than $\frac{1}{\alpha}$ of the maximum upper bound utility, for some constant $\alpha > 1$ that is chosen by the system, we can stop the algorithm (line 8–11).

Algorithm 7: Top-PEkExp(Q, I, g, k)

```
1  $\mathcal{L} \leftarrow$ partial materialized lattice structure;
2  while true do
3      $I_a \leftarrow$ getNextListRR();
4      $t \leftarrow I_a$.getNextItem();
5      if $Q \not\subseteq$ Kw($t$) then continue;
6      Generate necessary expansions using TopExp-Lazy;
7      $R^G \leftarrow$ GreedyMWIS($\mathcal{L}$);
8      $E_{topk} \leftarrow k$ expansions in $\mathcal{L}$ which have the largest upper
       bound utilities;
9      $U^* = sum_{e \in E_{topk}} \bar{u}(e)$;
10     if $u(R^G) \geq \frac{1}{\alpha} \times U^*$ then
11         return
```

It is clear that Algorithm Top-PEkExp can correctly return an α approximate answer for the maximum k path exclusive expansion problem. However, because we are using a greedy algorithm for calculating the k path exclusive expansions in each iteration, there may exist a better algorithm, e.g., which utilizes an exact algorithm, which can find an α approximate answer much earlier compared with our Top-PEkExp algorithm. This would trade more work done per iteration for achieving early termination. We leave a detailed study of optimal algorithms for the maximum k path exclusive expansion problem for future work.

5. EXPERIMENTS

In this section we will discuss the performance of our proposed algorithms. We use synthetic datasets to demonstrate the relative efficiency, scalability, and memory savings of various algorithms with respect to the naive algorithm. We also use a real dataset to demonstrate the quality of the expansions returned by our algorithms.

5.1 Experiment Setup and Data Sets

The goal of our experiments is two-fold: (i) Evaluate the efficiency and scalability of the algorithms proposed in this paper. (ii) Evaluate the quality of the expansions discovered by various algorithms. The experiments are done on a Xeon 2.5GHz Dual Core Windows 7 machine with 4GB RAM. All algorithms are implemented in Java using JDK/JRE 1.6.

We use two kinds of datasets in our experiments. First, we generated synthetic datasets to compare the performance of various algorithms with the naive algorithm. The metrics we used for the comparison include the running time, number of items accessed and number of expansions generated during the process. We generated 5 synthetic datasets with size from 8000 to 12000, and for all these datasets, attributes and keyword values are sampled from a power law distribution $x = e^{-\beta}$ with $\beta = 2$, which is in accordance with our observation in the real datasets.

The second dataset is a partially crawled dump of the ACM Digital Library. We obtained the items by combining result paper lists of 3 queries: (1) "xml" (2) "histogram" (3) "privacy". We chose these queries because they are all representative interesting research fields in the database community and also feature a good number of publications. The attributes we use for each paper are the average author publication number and the citation count. Values of both attributes are normalized into the range $[0, 1]$. For each paper, we extract its keywords from the title, keywords list and abstract. Stop words are removed and also stemming is done on all the keywords obtained. For each paper under consideration, we select the top 15 BM25 scored [20] keywords as its topic keywords. A manually created mapping table is used to map similar keywords into a common keyword, namely the most frequent among them. Note that some recent work on tag clustering/recommendation [22] can be leveraged to automate the creation of the mapping table.

After preprocessing, there are in total 48656 papers in the dataset, and 9000 distinct topic keywords. The quality of the expansions (keywords) generated is manually evaluated by domain experts consisting of grad students, who investigated the top expansions discovered, and checked whether the keywords make sense for the field.

5.2 Efficiency Study

First, the efficiency comparison of various algorithms on the synthetic datasets with fixed N and k is presented in Figure 4 A ($N = k = 10$). Note that for the path exclusive algorithm, we choose $\alpha = 0.1$ and $\alpha = 0.3$ as two settings for the algorithm. The running time, the number of items accessed and the number of expansions generated during execution of these algorithms are presented in Figure 4 A(1)-A(3) respectively.

For running time, both TopExp-Lazy and TopExp-LazyW (Weighted TopExp-Lazy) run much faster than the baseline TopExp-Naive algorithm. And the running time of PekExp based algorithms highly depend on the α parameter chosen.

For the number of items accessed, TopExp-LazyW, TopExp-Lazy and TopExp-Naive access roughly the same number of items. And for PekExp based algorithms, the number of items accessed may vary depending on the parameter α. With relatively small α, PekExp accesses significantly less items compared with other algorithms, hence much more efficient. Through our observation, for small α, PekExp usually stops very soon, and as we will illustrate in the next section, the quality of the expansions returned by PekExp based algorithms are also comparable compared with other algorithms. If the application needs immediate response in most cases and tolerable for occasion failure, PekExp would be a reasonable choice from the aspect of efficiency.

For the number of expansions generated, while accessing the same number of items, both TopExp-LazyW and TopExp-Lazy generate much less expansions than TopExp-Naive does. This could significantly lessen the space required and the related computations of generating and updating expansions. For TopExp-LazyW and TopExp-Lazy, the weighted algorithm takes more time and generates more expansions. This is because the weighted algorithm needs to keep different bounds for expansions which correspond to the same set of seen items but with different number of keywords. However, the space and time cost is still reasonable considering the flexibility we could achieve by customizing the weights.

In Figure 4 B(1)-B(3), we show efficiency comparison of the algorithms with k ranging from 10 to 15(number of items=10000, $N = 10$). With larger k, more information needs to be kept for the top expansions, so the updating cost is higher. Furthermore, because more items need to be accessed before the algorithm stop, the algorithm needs longer running time. This coincides with what shows in Figure 4 B(1)-B(2). The only exception is PekExp-alpha= 0.3, for this setting, the number of items accessed and the running time may decrease as k increases. This is because that as k grows, the upper bound of the top-k expansions will drop much faster as more low utility value expansions are included.

Similarly, in Figure 4 C(1)-C(3), we compare the algorithms with N ranging from 10 to 15(number of items=10000, $k = 10$). With larger N, because we need to keep track of more items to determine the bounds for each expansion, we could expect longer running time, more items to be accessed and more expansions to be generated.

We can conclude from these plots that the performance of our proposed algorithms are very robust with respect to different settings, and the performance of the algorithms grows linearly with respect to the size of the dataset, so these algorithms can easily scale to larger datasets. Furthermore, it is clear that the TopExp-Lazy algorithms (weighted and unweighted) always outperform the TopExp-Naive algorithm. PekExp based algorithms is an exception to these observations. With proper configuration it could give good performance, but generally the performance varies depending on the value selected for the parameter α.

5.3 Quality of the Generated Expansions

The quality of the expansions generated can be measured by the quality of the keywords of each top expansion. In Table 2, we show the top-20 expansions generated by the TopExp-Lazy and PekExp (shown as Path-exclusive in Table 2) algorithms on the ACM Digital Library dataset, as well as the corresponding keywords for the 3 selected queries for which expansions were generated. By manually analyzing the expansions generated, we can see that TopExp-Lazy works fairly well in generating expansions related to major subtopics of each query. E.g., for the query "histogram", according to the survey [17], the two most important applications of histogram techniques in databases have been "selectivity estimation" and "approximate query answering". For both topics we can find corresponding expansions in the list of expansions generated, e.g. "select", "approxim". Furthermore, Approximation is an established application of histograms, which constitutes a large portion

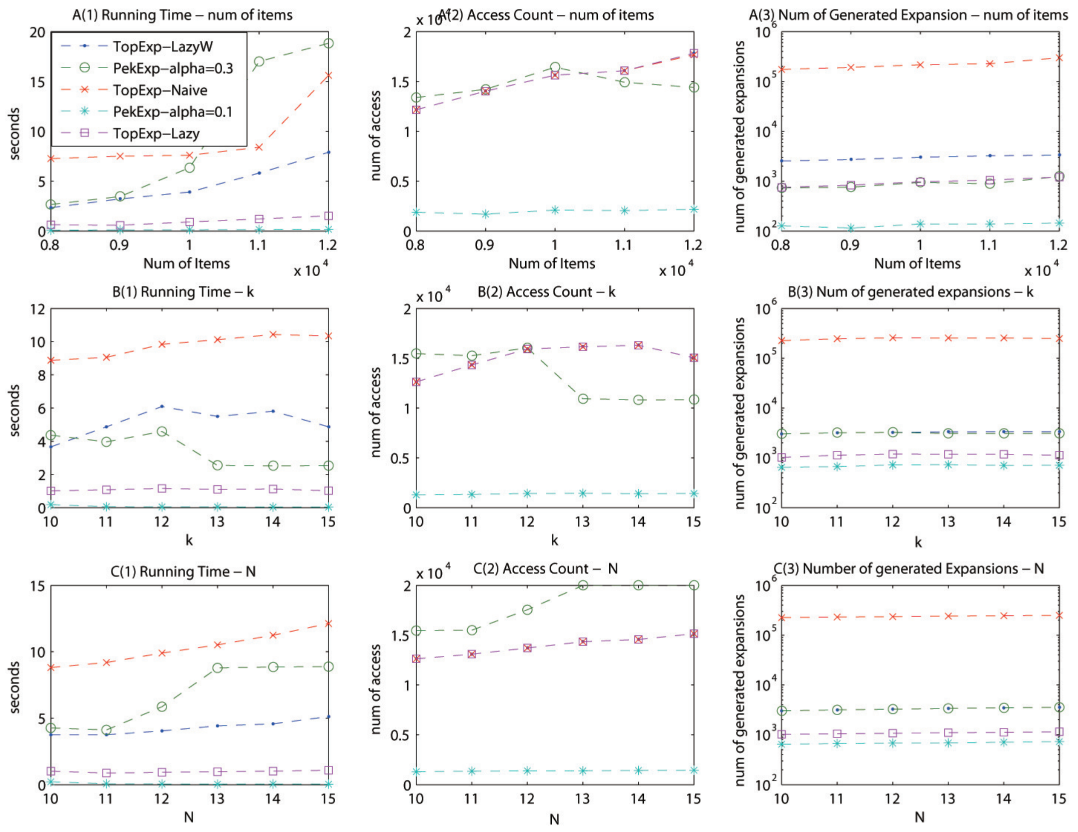


Figure 4: Efficiency experiments

of [17]. Indeed, accordingly we can see various low level expansions like "approxim wavelet", "approxim olap" and so on. Notice that other expansions of "histogram" also correspond to important aspects in the survey. For example, "multi-dimension" histogram is a relatively new and important subfield of histogram, and "optim" refers to an important application of approximation using histogram – query optimization. So in general, the expansions generated can cover most important aspects of the survey. Similar results can be observed for the other two queries as well.

While the quality of expansions returned by TopExp-Lazy are quite good, as can be found in Table 2, these expansions may contain redundant information. E.g., for query "histogram", the expansions returned include both the high level expansion "approxim" and low level (i.e., more refined) expansions "approxim wavelet", "approxim olap". Similarly, for query "privacy", we have "anonym" returned along with "anonym protect" and "anonym k", and for query "xml", we have "query" returned along with "query xpath" and "query store". Our path exclusive algorithm can avoid this problem by making sure that no expansion in the returned result set is a subset of other expansions in the result, thus avoiding such redundancy. By a manual inspection of the list of expansions in Table 2, it is easy to see that such redundancy is avoided by PekExp and also that the set of expansions returned by the that algorithm can also cover most of the important expansions for each query.

In sum, while both TopExp-Lazy and PekExp provide good quality results in terms of meaningful and most important expansions of queries on a real data set, as demonstrated by our experiments, PekExp has the added advantage that it can avoid redundancy in the returned expansions.

6. RELATED WORK

The area of top-k query processing has been studied extensively in the past several years [16]. Most of the top-k algorithms are based on the TA and NRA family of algorithms and their variants or enhancements [13]. The majority of them assume a monotone aggregation function for combining scores of items for different attributes. For each attribute, a non-increasing score-sorted list of items is maintained. In [11], Chakrabarti et al. consider the problem of finding top-k entities in a document corpus, where the score of an entity is defined as a weighted aggregation of the scores of its related documents. The proposed algorithm is similar to our naïve algorithm for the top-k expansion problem, and both algorithms are extensions of the NRA algorithm. However, the number of entities arising in [11] can be considered as a constant, whereas in our context, the number of expansions is exponential w.r.t. the total number of keywords, which can be huge. Thus, it is critical to generate the expansions as lazily as possible.

Our top-k expansion problem is also related to the recent efforts on faceted search [23, 15]. Li et al. [19] propose the problem of automatic generation of top-k facets for query results on Wikipedia. However, their work makes crucial use of a pre-defined category hierarchy in Wikipedia whereas we don't need to make such assumptions. In fact, as mentioned in the introduction, for the applications we consider, this assumption *cannot* be made.

Citation recommendation is another area where previous works [12, 9, 14, 12] often generate results as a ranked list of documents. By contrast, in our work we automatically group all the results into different expansions, and return to the user the top-k interesting expansions along with the relevant items.

histogram		**privacy**		**xml**	
TopExp-Lazy	**Path-Exclusive**	**TopExp-Lazy**	**Path-Exclusive**	**TopExp-Lazy**	**Path-Exclusive**
approxim	optimal	preserve mining	mining preserve	dtd conform	twig
bucket	approxim wavelet	anonym protect	anonym protect	twig holist	join
select	approxim olap	preserve	mining rule	query	ancestor-descend parent-child
optim	multi-dimension	mining	anonym mining	query xpath	conform
wavelet	approxim stream	anonym publish	individual	dtd	dtd tractable
approxim wavelet	approxim summary	anonym	perturb reconstruct	twig pattern	decide
approxim olap	frequency	anonym k	disclose disclosure	index	dissemin secure
multi-dimension	provable	protect individual	breach	access control	sql store
approxim stream	cost select	protect	mining reconstruct	holist pattern	typecheck
maintain	maintain remark	mining rule	access	dtd schema	node
approxim optim	alloc bucket	locat cloak	k	secure	encrypt secure
bucket multi-dimension	maintain size	access control	mining protect	index twig	control enforce
attribute	join select	mining anonym	control	query store	dynamic query
approxim construct	bucket workload-aware	anonym attack	location-based	twig	index subsequence
approxim summary	bucket reconstruct	anonym locate	categorize	query index	extension select
query	approxim near	locate	vertice	join	native timber
construct	predict range	individual	microdata quasi-identify	ancestor-descend parent-child	secure subject
sample	interval partition	disclosure	pose preserve	index holist	portion secure
frequency	build maintain	preserve anonym	anonym multi-dimension	conform	prevalence secure
stream summary	prohibit self-tuning	reconstruct perturb	scheme secret	holist	index indice

Table 2: Expansions generated by various algorithms.

In [21], the authors propose a way to facilitate the search process by suggesting interesting additional query terms. To measure the interestingness of an additional query term or keyword, they proposed the surprising score which is based on the co-occurrence of two keywords. Compared with our work, they don't consider the overlap between different sets of query keywords. Besides, their score function needs to access all items which are relevant to the query. Since this cannot be calculated at the query time in a scalable manner, they instead propose an approximate solution.

7. CONCLUSION

In this paper, we started with the observation that years after search engines first came into being, most current search services still return results as a flat ranked list of items, which is not ideal for users to easily get to the items they are really interested in. We studied the problem of how to better present search/query results to users. We considered a search scenario in which each item is annotated with a set of keywords and is equipped with a set of attributes, and proposed novel ways to automatically group query result items into different expansions of the query, corresponding to subsets of keywords. We proposed various efficient algorithms which can calculate top-k expansions, and we also studied additional desirable properties for the set of expansions returned, from a semantic perspective, whereby certain redundancies in the expansions returned can be avoided. With a detailed set of experiments, we not only demonstrated the performance of the proposed algorithms, we also validated the quality of the expansions returned by doing a study on a real data set. It is interesting to explore more desirable properties of the expansions returned, and to investigate more efficient algorithms which can return high quality expansion set and can handle the web scale.

8. ACKNOWLEDGEMENT

This work was supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

9. REFERENCES

[1] `http://scholar.google.com`.
[2] `http://www.bing.com`.
[3] `http://www.delicious.com`.
[4] `http://www.flickr.com`.
[5] `http://www.google.com`.
[6] `http://www.quora.com`.
[7] `http://www.yahoo.com`.
[8] Sihem Amer-Yahia. I am structured: Cluster me, don't just rank me. In *BEWEB*, 2011.
[9] Steven Bethard and Dan Jurafsky. Who should i cite: learning literature search models from citation behavior. In *CIKM*, pages 609–618, New York, NY, USA, 2010. ACM.
[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
[11] Kaushik Chakrabarti, Venkatesh Ganti, Jiawei Han, and Dong Xin. Ranking objects based on relationships. In *SIGMOD*, pages 371–382, New York, NY, USA, 2006. ACM.
[12] Michael D. Ekstrand, Praveen Kannan, James A. Stemper, John T. Butler, Joseph A. Konstan, and John T. Riedl. Automatically building research reading lists. In *ACM RecSys*, pages 159–166, New York, NY, USA, 2010. ACM.
[13] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.
[14] Qi He, Jian Pei, Daniel Kifer, Prasenjit Mitra, and Lee Giles. Context-aware citation recommendation. In *WWW*, pages 421–430, New York, NY, USA, 2010. ACM.
[15] Marti A. Hearst. Clustering versus faceted categories for information exploration. *Commun. ACM*, 49:59–61, April 2006.
[16] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
[17] Yannis Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30. VLDB Endowment, 2003.
[18] Akihisa Kako, Takao Ono, Tomio Hirata, and Magnús M. Halldórsson. Approximation algorithms for the weighted independent set problem. In *WG*, pages 341–350, 2005.
[19] Chengkai Li, Ning Yan, Senjuti Basu Roy, Lekhendro Lisham, and Gautam Das. Facetedpedia: Dynamic generation of query-dependent faceted interfaces for wikipedia. In *WWW*, pages 651–660, 2010.
[20] S. E. Robertson, S. Walker, and M. M. Hancock-Beaulieu. Okapi at trec-3. In *TREC-3*, pages 109–126. NIST, 1994.
[21] Nikos Sarkas, Nilesh Bansal, Gautam Das, and Nick Koudas. Measure-driven keyword-query expansion. *PVLDB*, 2(1):121–132, 2009.
[22] Yang Song, Ziming Zhuang, Huajing Li, Qiankun Zhao, Jia Li, Wang-Chien Lee, and C. Lee Giles. Real-time automatic tag recommendation. In *SIGIR*, pages 515–522, New York, NY, USA, 2008. ACM.
[23] Roelof van Zwol, Börkur Sigurbjornsson, Ramu Adapala, Lluis Garcia Pueyo, Abhinav Katiyar, Kaushal Kurapati, Mridul Muralidharan, Sudar Muthu, Vanessa Murdock, Polly Ng, Anand Ramani, Anuj Sahai, Sriram Thiru Sathish, Hari Vasudev, and Upendra Vuyyuru. Faceted exploration of image search results. In *WWW*, pages 961–970, New York, NY, USA, 2010. ACM.