Keyword Search for Data-Centric XML Collections with Long Text Fields

Arash Termehchy Marianne Winslett
Department of Computer Science, University of Illinois, Urbana, IL 61801
{termehch,winslett}@illinois.edu

ABSTRACT

Users who are unfamiliar with database query languages can search XML data sets using keyword queries. Current approaches for supporting such queries are either for text-centric XML, where the structure is very simple and long text fields predominate; or data-centric, where the structure is very rich. However, long text fields are becoming more common in data-centric XML, and existing approaches deliver relatively poor precision, recall, and ranking for such data sets. In this paper, we introduce an XML keyword search method that provides high precision, recall, and ranking quality for data-centric XML, even when long text fields are present. Our approach is based on a new group of structural relationships called **normalized term presence correlation** (NTPC). In a one-time setup phase, we compute the NTPCs for a representative DB instance, then use this information to rank candidate answers for all subsequent queries, based on each answer's structure. Our experiments with 65 user-supplied queries over two real-world XML data sets show that NTPC-based ranking is always as effective as the best previously available XML keyword search method for data-centric data sets, and provides better precision, recall, and ranking than previous approaches when long text fields are present. As the straightforward approach for computing NTPCs is too slow, we also present algorithms to compute NTPCs efficiently.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Designs, Performance

1. INTRODUCTION

The IR community has been developing retrieval techniques for *text-centric* XML [10], where structures are simple and structural information plays almost no role in retrieval. In *data-centric* XML, structure carries important information about objects and their relationships. Data-centric XML can support much more powerful query interfaces than unstructured data can, but non-expert users have no easy way to tap into that power with ad hoc queries. Following the success of keyword search interfaces in IR and Web search, researchers have proposed XML keyword search interfaces as a solution [5, 7, 8, 13, 10, 27, 23]. Since keyword queries do not explicitly specify the structural properties of the ideal answers, the underlying system must find the XML sub-structures that are relevant to the query, and preferably rank them based on their relevance. Further, the system should exploit XML structural information, to improve the system's precision, recall, and ranking quality.

As discussed in more detail in the next section, almost all XML keyword query approaches incorporate pruning and ranking heuristics based on shallow structural properties of the data (e.g., smallest answers are best). A previous user study involving 65 queries over a movie database and a bibliographic database [24] showed that these heuristics lead to many irrelevant answers (low precision), missing relevant answers (low recall), and poor or non-existent ranking of answers. To solve these problems, one can use a more powerful ranking approach, based on statistical measures of the correlation between schema elements [11, 9, 23]. Intuitively, answers that involve closely correlated schema elements should be ranked higher than less correlated answers. Such schema-correlation-based measures have been shown to be effective for ranking query answers for data-oriented XML without long text fields [23], but they do not handle long text fields such as paper abstracts or movie plot summaries appropriately. This type of data sets are proliferating due to the increasing popularity of constructing and querying annotated corpora [4].

In a nutshell, the problem is that these measures consider two field values to be completely different, even if they differ in only one word. For example, consider a movie database that includes information about writers, movie plot lines, and movie tag lines (lines from trailers). Each movie has its own unique tag lines and plot lines, so existing correlation-based measures find that writers' names are as highly correlated with tag lines as with plots. In practice, however, a writer's plots tend to involve similar situations or characters, while the tag lines of his or her shows have little similarity. Thus intuitively, writers should be more correlated with plots than with tag lines. Overgeneralizing slightly, any correlation measure that cannot capture this intuition will tend to rank a query answer that includes a writer and

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

a phrase from a tag line higher than an answer including a writer and a phrase from a plot line, which is undesirable in practice.

To address this problem, we introduce *term based correlation* measures as an effective way to capture our intuition about correlations between similar pieces of text, while at the same time exploiting all other available structural information in an XML database. Our contributions:

- We introduce a particular form of term based correlation called **normalized term presence correlation** (NTPC) and show how to incorporate NTPC into an XML query system. We show how to perform a one-time computation of NTPCs for schema elements, using a populated database instance. For all subsequent queries, the precomputed NTPCs are used to rank candidate answers based on their structure. NTPCs only need to be recomputed if structural changes introduce a new type of schema element. We also show how to combine NTPCs with traditional IR ranking methods, so that query answer rankings consider both content and structure.
- Through an extensive user study with two real-world data sets, we show that the precision, recall, and ranking quality of a NTPC-based approach to query answering is always at least as high that of as seven other previously proposed approaches to XML keyword search. We also show that NTPC provides better ranking than previous approaches when the database includes long fields.
- The straightforward approach to computing NTPCs is prohibitively slow for large data sets. We present novel optimizations that allow NTPCs to be computed for .5-1.1 GB data sets in .5-2.5 days, which is reasonable for a one-time setup cost.

In the reminder of the paper, Section 2 discusses current XML keyword search systems. Section 3 defines NTPC. Section 4 presents optimization techniques and algorithms for computing NTPC. Section 5 describes our query system, Section 6 presents empirical results, and Section 7 concludes the paper.

2. MOTIVATION

2.1 XML Patterns

We model an XML DB as a tree $T = (r, V, E, L, C, D)$, where V is the set of nodes in the tree, $r \in V$ is the root, E is the set of parent-child edges between members of V, $C \subset V$ is a subset of the leaf nodes of the tree called *content nodes*, L assigns a label to each member of $V - C$, and D assigns a data value (e.g., a string) to each content node. We assume no node has both leaf and non-leaf children, and each node has at most one leaf child; other settings can easily be transformed to this one. We also ignore all non-content leaf nodes, as they do not affect rankings. Each node can be identified by its *path* from the root; e.g., node 5 in Figure 1 has path *bib paper title*. Each *subtree* $S = (r_S, V_S, E_S, L_S, C_S, D_S)$ of T is a tree such that $V_S \subseteq V$, $E_S \subseteq E$, $L_S \subseteq L$, and $C_S \subseteq C$. A *keyword query* is a sequence $Q = t_1 \cdots t_q$ of terms. A subtree S is a *candidate answer* to Q iff its content nodes contain at least one instance of each term in Q. The root of a candidate answer is the *lowest common ancestor* (LCA) of its content nodes [19]. When no confusion is possible, we identify a candidate answer by its root's node number.

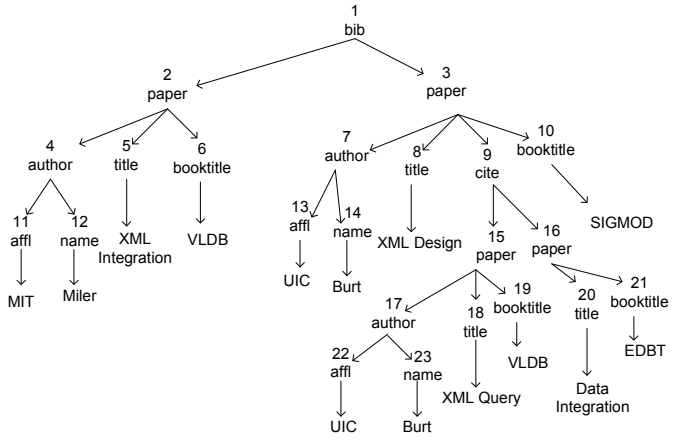

Figure 1: DBLP database fragment

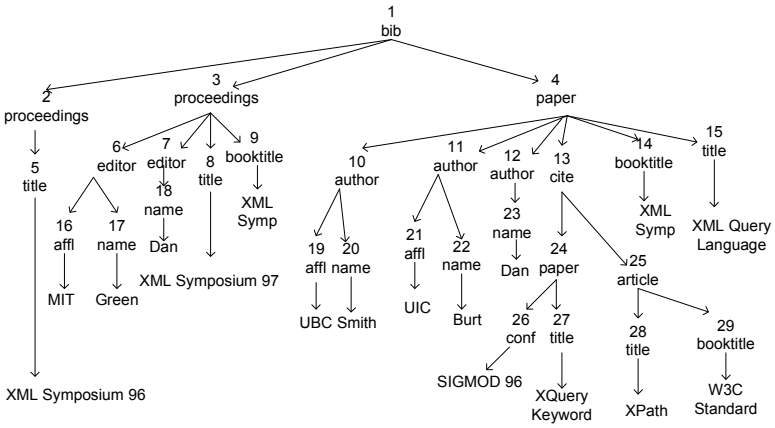

Figure 2: DBLP database fragment

Consider the depth-first traversal of a tree, where we visit the children of a node in the alphabetic order of their labels. Each time we visit a node, we output its number (or content); each time we move up one level in the tree, we output *-1*. The result is the unique *prefix string* for that tree [28]. For instance, the prefix string for the subtree rooted at node 4 in Fig. 1 is *4 11 MIT -1 -1 12 Miller -1 -1*. Trees T_1 and T_2 are **label isomorphic** if the nodes of T_1 can be mapped to the nodes of T_2 in such a way that node labels are preserved and the edges of T_1 are mapped to the edges of T_2. A **pattern** concisely represents a maximal set of isomorphic trees (its **instances**) [23]. The pattern can be obtained from any member of the set, by replacing each node number in its prefix string by the corresponding label. For instance, pattern *bib paper title -1 -1* corresponds to trees *1 2 5 -1 -1* and *1 3 8 -1 -1* in Fig. 1. The **value** of a subtree (if it exists) is the content associated with its leaves. For example, the value of *1 2 5 -1 6 -1 -1* in Fig. 1 is ("XML Integration", "VLDB"). The values of a pattern are all the values of its instances.

2.2 The State of the Art

The consensus in XML keyword search is that the best answers are the most specific subtrees containing the query terms [5, 7, 8, 13, 14, 27]. The specificity of a subtree depends on the strength of the relationship between its nodes.

For instance, if two nodes merely belong to the same bibliography, such as titles of two different papers, then the user will not gain any insight from seeing them together in an answer. If the nodes belong to the same paper, such as the paper's title and author, the user will surely benefit from seeing them together. If the nodes represent titles of two different papers cited by one paper, the answer might be slightly helpful.

The *baseline approach* for XML keyword search returns *every* candidate answer ([19], with modest refinements in [8]). For instance, consider the DBLP fragment from www.informatik.uni-trier.de/~ley/db in Fig. 1. The answer to query *Integration Miller* is (rooted at) node 2. The baseline approach has perfect recall but very low precision. For example, for query Q_1 = *Integration EDBT*, the baseline approach returns the desired answer of node 16, but also the unhelpful root node. In Q_2 = *Integration VLDB* for Fig. 1, candidate answers nodes 9 and 1 are not very helpful. The node 9 tree contains two otherwise-unrelated papers cited by the same paper, and the node 1 tree contains otherwise-unrelated papers in the same bibliography. A good approach should rank them below helpful answers, or omit them.

Current approaches to XML keyword search fall into several categories. The first category uses shallow structural heuristics to filter out irrelevant answers [27, 21, 15, 14, 13]. For instance, SLCA [27] eliminates candidate LCAs that are ancestors of any other candidate LCA. SLCA implicitly assumes that candidate answers deep in the XML tree are more relevant to the query than answers high up in the tree. For Q_1 in Fig. 1, SLCA does not return node 1, but SLCA does return node 9 for Q_2. Since SLCA does not rank its answers, the user gets a mix of unhelpful and desirable answers. Since SLCA filters out some candidate answers, its recall is less than the baseline approach. For instance, for query Q_3 = *XML Burt*, nodes 3 and 15 are desirable; but node 3 is an ancestor of node 15, so node 3 will not be returned. MSLCA and MaxMatch use heuristics similar to SLCA's [21, 15], and share SLCA's deficiencies.

VLCA [13] eliminates candidate answers where two non-leaf nodes have the same label. The idea is that non-leaf nodes are instances of the same entity type if they have duplicate labels (DLs), and there is no interesting relationship between entities of the same type. We refer to this heuristic as *DL*. For instance, the subtree rooted at node 9 does not represent a meaningful relationship between nodes 19 and 20, because they have the same label and type. Therefore, node 9 should not be an answer to Q_2.

DL is not an ideal way to detect nodes of similar type. For example, nodes *article* and *paper* in Fig. 2 have different names but represent similar objects. As a result, for the query Q_4 = *SIGMOD XPath*, DL returns node 11, which is undesirable. DL cannot detect uninteresting relationships between nodes of different types, either; it does not filter out node 1 for query Q_5 = *UBC Green* in Fig. 2. Further, sometimes there *are* meaningful relationships between similar nodes, even in a DB with few entity types. For example, DL does not return any answer for *Smith Burt* in Fig. 2, as it filters out node 4. Like SLCA, VLCA does not rank its candidate answers. MLCA [14] uses a quite similar approach. A more detailed comparison of the methods in this group can be found elsewhere [24].

The second category of XML keyword search approaches combines IR-based techniques with the aforementioned structural heuristics to rank candidate answers [5, 1]. XSearch [5] filters out candidate answers using DL heuristics and ranks the remainder using their TF/IDF score and the number of nodes they contain (their size). However, size is an unreliable predictor of relevance. Consider query *SIGMOD VLDB* for Fig. 1. The answers rooted at nodes 1 and 3 have the same size, but only the one at node 3 is relevant. TF/IDF methods do not consider XML structure, so they cannot remedy this deficiency or the issues with DL.

XReal [1] filters out entity types that do not contain many of the query terms. Then it ranks subtrees higher if they and their entities' types have more of the query terms. For instance, DBLP has few books about *Data Mining*, so XReal filters out all *book* entities when answering the query *Data Mining Han* – even Han's textbook. XReal does not consider the relationship *between* nodes of a subtree when it ranks its answers, so irrelevant subtrees can be ranked very high. For query *SIGMOD 1997*, XReal ranks the subtree rooted at node 4 in Fig. 2 higher than the 1997 papers with *booktitle* SIGMOD. This is because *SIGMOD* occurs very frequently in subtrees rooted at *cite*. Even if we ignore the importance of the *cite* entity type in XReal ranking, XReal still ranks papers published in 1997 below papers that cite multiple articles and papers published in 1997. Generally, IR ranking techniques do not exploit XML's structural properties.

XRank [7] finds LCAs using a modified baseline approach, and uses a PageRank-based approach to rank subtrees. It has the same precision problems as the baseline approach. Also, PageRank is effective only in certain domains and relationships and is not intended for ranking trees; e.g., all the *Proceedings* tags in Fig. 2 have the same PageRank.

CR [23] ranks candidate answers based on the strength of the correlation between the nodes in their patterns. Patterns with high correlation represent strong relationships. For example, consider Q_2 in Fig. 1. All papers in DBLP are associated through the DBLP root, so the correlation between the values of the pattern *bib paper booktitle -1 -1 paper cite paper title -1 -1 -1 -1* is very low in the full DBLP. Therefore, CR ranks candidate answers rooted at the DBLP root last, or omits them. Pattern *cite paper title -1 -1 paper booktitle -1 -1* represents a more meaningful relationship, as the correlation between its values is higher than for the previous pattern. However, two papers cited by the same paper do not always have a meaningful relationship. Therefore, CR ranks the instance of this pattern rooted at node 9 low. The correlation between the values of the pattern *paper title -1 booktitle -1* is higher yet, as each paper appears in only one conference. Therefore, CR ranks the candidate answer rooted at node 2 first.

The size of a pattern is not related to its correlation. For instance, *bib paper title -1 -1 paper title -1 -1* and *cite paper title -1 -1 paper title -1 -1* in Fig. 1 have the same size. But the correlation of the first is less than that of the second, as *every* two papers are connected to each other through the root of the subtree. The same goes for *paper title -1 author name -1 -1* and *paper booktitle -1 author name -1 -1* in Fig. 1. Since many authors publish their papers at the same conference and each title has only a few authors, the values of the first pattern are more correlated than the values of the second one.

CR computes correlations using a measure based on the concept of mutual information, called Normalized Total Correlation (NTC). CR has been combined with IR-style mea-

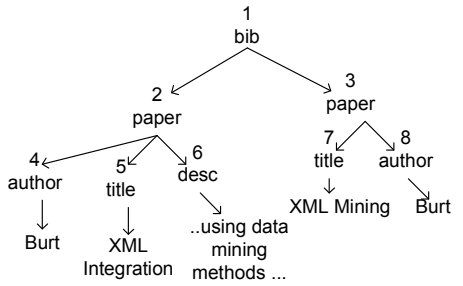

Figure 3: Citeseer database fragment

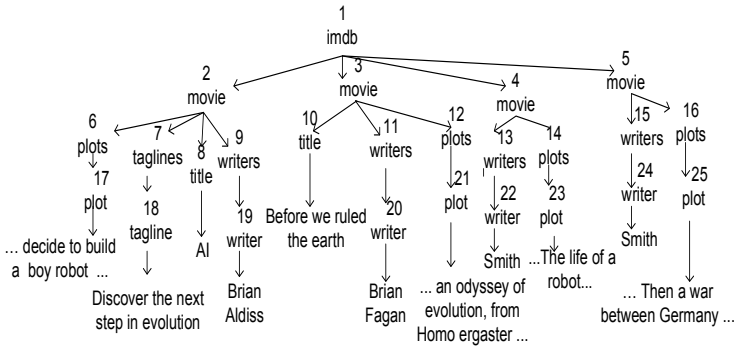

Figure 4: IMDB database fragment

sures to deliver better rankings [23]. As CR does not rely on shallow structural properties, it does not have the precision pitfalls mentioned for previous approaches. As CR does not remove candidate answers, it has better recall than the approaches that prune out answers. However, neither CR nor any of the other approaches discussed above can rank answers with long text fields well. We explore this problem in the next section.

2.3 Long Text Field Issues

Consider the Citeseer fragment from citeseer.ist.psu.edu/ in Fig. 3. The best answer to query *Burt Mining* is rooted at node 3. Since *desc* is a long text field, two different papers will almost never have the same description. However, a few papers have similar titles but different authors. Thus, the NTC of the values of *paper author -1 title -1* is less than that of *paper author -1 desc -1*, and CR inappropriately returns node 2 as the best answer.

IR-style heuristics and penalties for long fields will not solve this problem. For instance, consider the IMDB fragment from www.imdb.com in Fig. 4. Because tag lines are less indicative of a movie's content than plot lines, the best answer to query *Evolution Brian* is the subtree rooted at node 3. Different movies have different plot lines and tag lines. In the original IMDB, on average each movie has more *plot* nodes than *tagline* ((The famous sentences in the movies' trailers) nodes, so the NTC of *movie taglines tagline -1 -1 writers write -1 -1* is 1.48 and the NTC of *movie plots plot -1 -1 writers write -1 -1* is only 1.37. Thus, CR ranks node 2 above node 3. As the *plot* field is longer than the *tagline* field, penalizing long fields will not solve the problem.

The problem can occur for short text fields as well. Consider query *XML Dan* for Fig. 2. The best answer is at node 4, which gives papers about XML written by Dan. The node 3 answer should be ranked second, as it is less interesting for most users. But since each proceedings has a different title, the NTC of *proceedings title -1 editor name -1 -1* is the same as the NTC of *paper title -1 author name -1 -1* in this fragment. NTC does not consider the fact that the proceedings titles for nodes 2 and 3 are almost identical; if it did, the correlation of the first pattern would drop and we would get the desired ranking.

Thus, intuitively, we should consider the individual components (words) of each value when computing correlations, both for long and short fields. For instance, the words in movie tag lines are not as representative of the movie's subject as the words in its plot lines. Thus in Fig. 4, the terms in the values of the field *writer* are more correlated with those of *plot* than *tagline*.

Correlation mining is an active research area in data mining and databases [3, 11, 16]. But most research has focused on finding the correlations between data items themselves, rather than computing the collective correlation between their columns or nodes. Even the systems in the latter category [9, 11, 12] consider the values of each column as a whole, so we cannot apply them to our ranking problem.

3. TERM BASED CORRELATION

3.1 Preliminaries

A **root-subtree** is a subtree whose root is the root of the XML DB and whose leaves are the parents of content nodes [23]. For instance, *1 2 5 -1 6 -1 -1* is a root-subtree in Fig. 1. If the root-subtree is a path, we call it a **root-path**. Each root-subtree contains at least one root-path. Every maximal set of isomorphic root-subtrees in a tree T corresponds to a pattern. The **size of a root-subtree pattern** is the number of root-path patterns it contains. From now on, the only patterns we consider are those of root-subtrees.

Definition 1. $W(r_1 : w_1, \ldots, r_n : w_n)$ is a **term** of root-subtree T containing root-paths $r_1, \ldots, r_n$, with **value** $(r_1 : v_1, \ldots, r_n : v_n)$, if $w_1, \ldots, w_n$ are non-stop words that occur in values $v_1, \ldots, v_n$, respectively.

The **terms** of pattern p containing root-path patterns $(p_1, \ldots, p_n)$ are the union of the sets of terms of its instances. We write p's terms as $W(p_1 : w_1, \ldots, p_n : w_n)$, reflecting which root-path pattern contains which term. For instance, *(p_1 : Design)* is a term of the root-path pattern *bib paper title -1 -1* and p_1 : *Design*, p_2 : *SIGMOD* is a term of the root-subtree pattern t_1 = *bib paper title -1 booktitle -1 -1* in Fig. 1.

Each term $W(p_1 : w_1, \ldots, p_n : w_n)$ is associated with 2^n possible **events**. Each event takes the form $E(p_1 : f(w_1), \ldots, p_n : f(w_n))$, where each $f(w_i)$ is either w_i or $\bar{w_i}$, depending on whether w_i does or does not occur in p_i.

Definition 2. The **occurrence probability** of an event $E(p_1 : f(w_1), \ldots, p_n : f(w_n))$ for term W in a root-subtree pattern q is $O(E) = \frac{|t(E)|}{|t(q)|}$, where $|t(E)|$ is the number of terms where p_i contains w_i, if $f(w_i) = w_i$; or where p_i does not contain w_i, if $f(w_i) = \bar{w_i}$ $(1 \leq i \leq n)$. $|t(q)|$ is the total number of terms of q in the DB.

In Fig. 1, if q = *bib paper title -1 -1*, $O(p_1$: "Design") $= \frac{1}{4}$, as this pattern has 4 terms.

Definition 3. The **presence probability** of an event $E(p_1 : f(w_1), \ldots, p_n : f(w_n))$ for term W in a root-subtree pattern

q is $P(W) = \frac{|E|}{|q|}$, where $|E|$ is the number of instances of q where p_i contains w_i, if $f(w_i) = w_i$; or p_i does not contain w_i, if $f(w_i) = \bar{w_i}$ $(1 \leq i \leq n)$. $|q|$ is the number of instances of q in the DB.

In Fig. 1, if q = *bib paper title -1 -1*, $P(p_1 : \text{Design}) = \frac{1}{2}$, as the pattern has two instances and "Design" occurs in one.

A root-subtree pattern represents the joint distribution of the root-paths it contains. For instance, the root-subtree pattern t_1 = *bib paper title -1 booktitle -1 -1* represents an association between root-path patterns p_1 = *bib paper title -1 -1* and p_2 = *bib paper booktitle -1 -1*.

Intuitively, the entropy of a random variable indicates how predictable it is [6]. We define the entropy of term W:

Definition 4. Given term W of pattern p of size n whose events $E_1, \ldots E_{2^n}$ have presence probability $P(E_1), \ldots, P(E_{2^n})$ respectively, the **presence entropy** of p is
$H_p(W) = \sum_{1 \leq i \leq 2^n} P(E_i) \lg (1/P(E_i))$.

For instance, consider the term *(p_1 : robot, p_2 : Aldis)* in the pattern *imdb movie plots plot -1 -1 writers writer -1 -1 -1*, where p_1 is *imdb movie plots plot -1 -1 -1* and p_2 is *imdb movie writers writer -1 -1 -1* in Fig 4. The term has four events. Considering only DB fragment in the figure, we can compute their presence probabilities to find the the presence entropy for the term: $2 * 1/4 \lg (4) + 2/4 \lg (2) = 1.5$.

The **occurrence entropy** $H_o(W)$ of term W can be defined similarly.

Definition 5. Given the pattern p containing terms $W_1, \ldots, W_N$ with occurrence probabilities $O(W_1), \ldots, O(W_N)$ respectively, the **collective entropy** of p is
$H(p) = \sum_{1 \leq i \leq N} O(W_i) \lg (1/O(W_i))$.

As opposed to the presence and occurrence entropies, collective entropy is defined over all terms of a pattern. Notice that we cannot define the collective entropy based on presence probability, as the sum of the presence probabilities of the terms of a pattern could exceed one. If the pattern has more than one path, we also call its entropy *joint entropy*, as it is defined over a joint distribution.

3.2 Correlation Measures

Total correlation [25] is closely related to mutual information [6]; it measures the correlation between random variables. It is defined over random variables $A_1, \ldots, A_n$ as

$$I = \sum_{1 \leq i \leq n} H(A_i) - H(A_1, \ldots, A_n). \quad (1)$$

The greater the value of I, the more correlated the variables are. If the variables are independent, the value of I will be zero. Since each w_i of term $W(p_1 : w_1, \ldots, p_n : w_n)$ is a term itself, we can extend the definition of total correlation as follows:

Definition 6. The **total presence correlation** (TPC) of term $W(p_1 : w_1, \ldots, p_n : w_n)$ of pattern q is:

$$I_p(W) = \sum_{1 \leq i \leq n} H_p(w_i) - H_p(W). \quad (2)$$

In the same setting as the last example, the TPC of term *(p_1 : evolution, p_2 : Fagan)* is 0.61 and the TPC of term *(p_1 : robot, p_2 : Aldiss)* is 0.31. In this fragment, from the writer's name *Fagan*, we can predict that the movie is about *evolution* and vice versa. However, knowing the movie is about *robot* does not necessarily mean that its writer is *Aldiss*, so they are not as correlated as the previous term. Thus, the TPC of a term reflects the correlation between its components. In the above definition, the instances of path p_i in the DB are a superset of its instances that are subtrees of the instances of q. Thus, we compute $H_p(w_i)$ considering only the instances of p_i that are subtrees of the instances of q. The **total occurrence correlation** (TOC) of a term can be defined similarly.

We can measure the correlation of a pattern by averaging the sum of TPCs for all terms in the pattern. However, there are many weakly correlated terms in patterns. For instance, the original IMDB has many terms in the title, tag line, or plot that are not correlated or are weakly correlated with the terms in the name of the movie writers or the terms in other fields. Such words usually have high frequency in the DB. Hence, patterns that intuitively should have different correlations may look very similar if we average over all their terms. To prevent this problem, we average over the top-k correlated terms, where k is reasonably large. If k is too small, we face the same problem, as there are always some terms that are highly correlated in most patterns. Through empirical evaluation, we found that a value of 50-100 is appropriate for a large database such as DBLP or IMDB.

TOC and TPC both measure the collective correlation of a pattern. However, their different properties make them appropriate for different applications. Consider the term $W(p_1 : w_1, p_2 : w_2)$ in pattern q, where the values of p_2 are relatively long. With TOC, w_1 is associated with many terms in p_2 where one of them is w_2. Thus, TOC finds the term W to be relatively weakly correlated. Since TPC views the presence of the components of the term W in an instance of p as an association between them, TPC gives W a higher correlation. As mentioned before, a good correlation measure must work equally well for long and short fields. Hence, we choose TPC for our application, keyword search.

Among patterns of size 1, those with more variety are more helpful for users. For instance, in Fig. 1, *title* is more helpful for most users and queries than *booktitle*. Thus if both match the input query, the system should rank *title* first. Since the terms of highly repetitive fields such as *booktitle* are quite evenly distributed in the database and there are so few of them, averaging over their top-k entropies returns a relatively high value that does not reflect the true variety of their information. Therefore, we use collective entropy from Definition 5 to rank patterns of size one.

Since users prefer smaller patterns, we penalize larger patterns in the final formula for TPC:

$$NTPC(W) = g(n) \times \frac{TPC(W)}{H(W)}, n > 1. \quad (3)$$

$g(n)$ is a function that penalizes larger patterns; $g(n) = n^2/(n-1)^2$ performs well in practice [23]. Exploring options for $g(n)$ is an interesting area for future work.

The values of NTPC and collective entropy for all patterns in the DB should be computed in a separate phase before the first queries are submitted to the system. If the DB does not undergo drastic structural changes that introduce new node types and patterns, this computation need never be repeated. For example, the tightness of the relationship between paper titles and their authors will not change, no

matter how many years go by or new conferences are introduced. However, if a new field *paper-body* is introduced, we must compute the NTPCs for candidate answers that include that new field.

To answer a query, we find all candidate answers and their associated patterns, and look up those patterns in a table of precomputed NTPCs. We rank answers according to their NTPC values. We choose to omit answers with zero NTPC, as they are especially irrelevant.

NTPC-based ranking successfully handles all the examples described earlier. For instance, the NTPC of *proceedings title -1 editor name -1 -1* in the full DBLP is 1.41, and the NTPC of *paper title -1 author name -1 -1* is 1.73. In the full IMDB, the NTPC of *movie taglines tagline -1 -1 writers write -1 -1* is 1.25, while the NTPC of *movie plots plot -1 -1 writers write -1 -1* is 1.49. Section 6 contains a detailed evaluation of the effectiveness of NTPC-based ranking.

4. COMPUTING NTPC

4.1 Optimization Techniques

Researchers have experimented with techniques for efficiently computing correlations. Unfortunately, these techniques rely on domain characteristics that are not applicable for keyword search over XML with long fields. For example, in keyword search, many strongly correlated terms are not frequent, so we cannot consider only frequent terms [3]. For keyword search, the minimum interesting value of NTPC is too small for it to be helpful as a cutoff during correlation computation [18]. Nor is the number of distinct values for a field necessarily much less than its total number of values, which would allow us to use sampling [9]. Nor can we adopt the techniques for precomputing NTC quickly [23], as NTC computes the correlation for entire values, not for the terms of a pattern. However, as in [23], we can assume that the maximum number of keywords in a query is relatively low (2.5 on average according to IR studies [26]). Thus, the size of the patterns we seek does have a domain-dependent upper bound $MCAS$ (maximum candidate answer size). For instance, empirical studies suggest that 4 is a reasonable $MCAS$ value for bibliographic DBs [23].

The following lemma reduces the number of patterns for which we must compute NTPC. We call two events **presence independent** if their presence probabilities are independent of one another.

LEMMA 1. *Consider the term $W(p_1 : w_1, \ldots, p_n : w_n)$, $n > 1$, of the pattern q, with events $E(p_1 : f(w_1), \ldots, p_n : f(w_n))$. If the root of q is the DB root, then all components $p_1 : f(w_1), \ldots, p_n : f(w_n)$ are presence independent.*

PROOF. We show the property for two arbitrary components of the term itself. The proof is similar for other numbers of components and other events. For every $1 \leq i, j \leq n$ we have:

$$\begin{aligned} P(p_i : w_i | p_j : w_j) &= \frac{P(p_i : w_i \cup p_j : w_j)}{P(p_j : w_j)} \\ &= \frac{|w_i||w_j|/|p_i||p_j|}{|w_j|/|p_j|} \\ &= \frac{|w_i|}{|p_i|} \\ &= P(p_i : w_i). \end{aligned} \tag{4}$$

□

We can view the components of a term's events as a random variable that assumes value 0 when $f(w_i) = w_i$ and 1 when $f(w_i) = \bar{w_i}$. Since all the events of these random variables are independent, they are independent, too. Therefore, we have:

COROLLARY 1. *The NTPC of patterns rooted at the root of the database is zero.*

Hence, when computing NTPCs we ignore all patterns rooted at the root of the database tree.

The number of pattern instances containing a term is the **frequency** of the term. From the properties of total correlation, it follows that infrequent terms have relatively low NTPC. The frequency of term $W(p_1 : w_1, \ldots, p_n : w_n)$ is less than the frequencies of its components $w_i, \ldots, w_n$. As explained later in this section, we compute the NTPC of a pattern using the information from its root-path patterns. Before computing the NTPC of patterns of size $n > 1$, we remove all terms in root-path patterns p whose frequencies are less than $\epsilon|p|$, where $0 < \epsilon < 1$. Similarly, for terms of very high frequency, we use an upper frequency cutoff threshold $1 - \epsilon$, and remove components with relative frequency above that limit. The appropriate value of ϵ depends on the number of root-path instances in the DB. In our experiments we used $\epsilon = 0.01$, with two exceptions. First, a root-path pattern will have members with very low frequency if the entity represented by the root-path pattern is a key or semi-key and each value of the pattern has only one or two terms, such as for ISBN numbers in a bibliographical database. We do not remove any term from such patterns, as that would not leave any term in the pattern. Some entities have terms that are very frequent, such as *payment-methods* with values *cash, check, credit card*. We do not remove any terms from such patterns, either.

4.2 NTPC Computation Algorithms

The ComputeNTPC algorithm in Fig. 5 gives an overview of how to compute NTPC and collective entropy. First, ComputeNTPC reads the XML data set in a depth first manner, finds the root-path patterns, and creates a **compressed index** (CI) for each root-path pattern. Each entry in a CI contains a root-path instance and the terms in its value. We define a 32 bit key for each term, and store the key instead of the term in the index. We do not need the terms to compute the collective entropies and NTPCs. Each root-path instance is represented in the CIs by the Dewey code [22] of its leaf node. Every Dewey code is stored in a bitmap to save space. These optimizations reduce the space requirements and enable ComputeNTPC to keep the CIs in main memory. For instance, for the roughly 1 GB IMDB data set, the average CI size was 4MB. As we use only one instance of CI for each root-path pattern throughout ComputeNTPC, this ensures modest space overhead and drastically reduces run time. The root-path instances in each CI entry are sorted according to their depth-first traversal order in the DB.

The next step in computing NTPCs is to find all patterns in the DB. We primarily use techniques from previous work [23] to generate these patterns efficiently, and discuss only the differences here. After creating CIs, ComputeNTPC computes the collective entropy for each root-path pattern

Input: XML data file *data*
Input: Maximum size $MCAS$ of patterns to compute NTPC for
Input: Minimum term frequency ϵ
Output: Table CT of NTPC for *data*

```
   /* Find the root-paths and build their CIs    */
1  invdx = Create_CI(data);
   /* Compute the collective entropies           */
2  forall p ∈ invdx do
3      clt_Entropy(p);
       /* Prune frequent and infrequent terms    */
4      prune(p,ϵ);
   /* Initialize the set of prefix classes       */
5  pfxSet ← ∅;
   /* Add all root-path patterns as one prefix
      class                                      */
6  pfxSet.add(invdx);
7  for k = 2 to MCAS do
8      nextPfxSet ← ∅;
9      last = ();
10     forall pfx ∈ pfxSet do
11         forall p ∈ pfx do
               /* Compute all prefix classes with
                  prefix p                       */
12             nextPfx ← ∅;
13             forall q ∈ pfx do
14                 Jnt ← join_Pattern(p, q);
15                 forall r ∈ Jnt do
16                     if subTrees(r) ⊄ pfxSet then
17                         continue;
                       /* Find the root-paths and join
                          levels for the new pattern */
18                     rp ← rootPaths(r);
19                     jl ← joinLevels(r);
                       /* Join the CIs           */
20                     jTable ← join_CIs(rp,jl);
21                     if jTable.freq = 0 then
22                         continue;
                       /* Compute the NTPC       */
23                     CT[r] ← NTPC(jTable);
24                     if k ≠ MCAS then
25                         nextPfx.add(r);
26             if k ≠ MCAS then
27                 nextPfxSet.add(nextPfx);
28     pfxSet ← nextPfxSet
29 return CT;
```

Figure 5: ComputeNTPC: algorithm to compute NTPCs

in line 3. In line 4, ComputeNTPC prunes terms whose relative frequencies are less than ϵ or more than $1-\epsilon$. In addition to lines 1-4, ComputeNTPC is different from the algorithms of [23] in its join operation at lines 18-20 and computing the correlation at line 23. After generating a pattern, in line 18 ComputeNTPC finds its root-paths, and then finds the levels of the LCAs of the root-paths. We call these levels **join**

Input: List *rp* of root-path patterns to join
Input: List *jl* of join levels
Output: Join table *jTable*

```
   /* root-paths to CI mapping                   */
1  path2Ind = pathIndexMap(rp);
   /* Distinct CIs                               */
2  indcs = indexes(path2Ind);
   /* Group levels for each root-path            */
3  gLvs = group_Levels(rp,jl);
4  for i = 0 to indcs.size - 1 do
5      nextInd.add(i);
6  repeat
       /* Read the terms from the CI entries     */
7      for i = 0 to indcs.size - 1 do
8          if i ∉ nextInd then
9              continue;
10         buf[i].clear();
11         buf[i].add(nextBuf[i]);
12         nextBuf[i] = NULL;
13         repeat
14             ent = indcs[i].next();
               /* No more entries                */
15             if ent = NULL then
16                 break;
               /* If entries of the same index can be
                  grouped                        */
17             if grp(buf[i][0],ent,gLvs[i]) then
18                 buf[i].add(ent);
19             else
20                 nextBuf[i] = ent;
21                 break;
22         until false ;
       /* Join groups of nodes                   */
23     for i = 0 to buf[0].size - 1 do
24         resBuf[i][0] ← buf[0][i];
25     for i = 1 to rp.size - 1 do
26         forall rent ∈ resBuf do
27             forall ent ∈ buf[path2Ind[i]] do
28                 if !eql(ent,rent) and
                   jin(ent,rent.last,jl[i-1]) then
29                     tmp.add(rent,ent);
30         resBuf ← tmp;
31         tmp.clear();
       /* Update the join table                  */
32     jTable.add(resBuf);
       /* Find the next indexes to read          */
33     nextInd ← nextGrp();
34 until !nextInd.empty() ;
35 return jTable;
```

Figure 6: JoinCIs: Algorithm to join compressed inverted indexes

```
   Output: The entries of the next groups
1  for i = 0 to nextInd.size-1 do
2      if nextBuf = NULL then
3          nextInd.clear();
4          return;
5  nextInd.clear();
6  if resBuf.size > 0 then
       /* move over all CIs                          */
7      for i = 0 to indcs.size-1 do
8          nextInd[i] = i;
9  else
       /* Find the groups with the smallest LCAs     */
10     node ← MAX_CODE;
11     for i = 0 to indcs.size-1 do
12         if grpLCA(buf[i][0],gLvs[i]) < node then
13             nextInd.clear();
14             nextInd.add(i);
15             node = buf[i][0];
16         else if grpLCA(buf[i][0]) = node then
17             nextInd.add(i);
```

Figure 7: NextGrp: algorithm to find the next group to join

```
   Input: Nodes n and m to group
   Input: Join level l
   Output: true if n and m can be grouped
1  if n = NULL or n.ancestor(l) = m.ancestor(l) then
2      return true;
3  return false;
```

Figure 8: Grp: algorithm for the grouping test

```
   Input: Nodes n and m to join
   Input: Join level l
   Output: true if n and m can be joined
1  if n.ancestor(l) = m.ancestor(l) and n.ancestor(l + 1)
   != m.ancestor(l + 1) then
2      return true;
3  return false;
```

Figure 9: Jin: test whether nodes can be joined at a particular level

levels. Each entry in a CI represents an instance of a root-path p_i containing a root-path term w_i. Thus, joining the entries of root-paths $p_1, \ldots, p_n$ of pattern q produces terms such as $W(p_1 : w_1, \ldots, p_n : w_n)$ of the pattern q.

Fig. 6 shows the join algorithm, JoinCIs. Many patterns have duplicates of the same root-path. For instance, the pattern *bib paper author name -1 -1 author name -1 -1 -1* has the root-path *bib paper author name -1 -1 -1* two times. These root-paths share the same CI and our goal is to use only one CI for each root-path. Thus, the join algorithm finds the mapping from root-paths to unique CIs in line 1. The loop from line 6-34 moves down the entries of the CIs and joins them if possible. For example, assume JoinCIs wants to join the instances of root-path p_i with root-path p_j at level l. An instance of p_i can join with an instance of p_j if they have a common ancestor at level l or above. For instance, we can join the root-path *bib paper author name -1 -1 -1* with the root-path *bib proceedings title -1 -1* at level 0 to create instances of the pattern *bib paper author name -1 -1 proceedings title -1 -1* in Fig. 2. The path instances *1 4 10 20 -1 -1 -1* and *1 4 11 22 -1 -1 -1*, whose LCA (node 4) is at level 1, can both join with the instance *1 3 8 -1 -1*. We call all instances of a root-path that have an LCA at level l or lower a **group**.

The Grp function shown in Fig. 8 checks whether two instances are in the same group. Since each root-path in a pattern is adjacent to two other root-paths (except the first and the last root-paths in the traversal of the pattern, which are only adjacent to one), the root-path has two candidate group levels. As the groups of this root-path must join to the groups of all its adjacent root-paths, the pseudocode chooses the lowest level of the two as the grouping level. Consider a pattern q that consists of root-paths p_i, p_j, and p_i, with join levels l_1 and l_2, where $l_1 < l_2$. To perform the join, we must group all instances of p_i at l_1 to form group g_1, and group its instances at level l_2 to form group g_2. However, according to the definition of a group, g_2 is a subset of g_1. Thus, we need only the group at level g_1. We use this technique to make the join operation faster. If a root-path occurs more than once in the same pattern, we create only one group for its instances. The group levels are computed at line 3 of JoinCIs. Every instance of a root-path belongs to only one group in each join operation. Thus, JoinCIs reads each member of a CI entry only once, and finds its group at lines 13-22.

JoinCIs then joins the groups in lines 25-31. The Jin method in Fig. 9 checks if two root-path instances can join at a given level. When two root-paths are the same, JoinCIs performs a self-join between the members of a group. Line 28 checks whether a root-path instance is different from the other root-path instances in a joint pattern instance. After joining the groups of the root-paths, JoinCIs inserts the keys of the joint terms in the **join table** at line 32. The join table is a hash table that maps the terms W of the produced pattern to the number of times $|W|$ they appear in the pattern. Line 33 finds the CIs whose groups are the next ones to read. The NextGrp algorithm in Fig. 7 shows how to find these CIs. Every root-path instance belongs to only one group. Also, the entries in each CI are sorted. Thus, if the join operation was successful (lines 6-8), NextGrp must advance the cursor over all CIs. Otherwise, it must advance the cursor over the groups whose LCAs have the smallest Dewey codes among the current groups (lines 11-17). For the reasons mentioned earlier, these groups cannot join with any other group. Lines 1-4 of NextGrp check whether we have finished with any CIs. The time complexity of joining CIs is linear in the number of groups in each CI. If the join operation produces at least one pattern instance, line 22 of ComputeNTPCs computes the NTPCs using the join table.

5. USING NTPC AT QUERY TIME

We keep the collective entropy and NTPC for each pattern in a hash table in main memory during query processing. Our query processing system, *SA3*, finds each candi-

	$\epsilon = 0$	$\epsilon = 0.001$	$\epsilon = 0.01$	$\epsilon = 0.02$	$\epsilon = 0.03$
DBLP	20.3	17.9	13.7	12.1	
IMDB	43.5	40.8	34.9	33.0	31.8
XMark	56.5	48.5	41.7	35.0	31.6

Table 1: NTPC computation time in hours, for different choices of ϵ

	NTPC	CR	XSearch	XReal	PN	XRank
IMDB	0.701	0.510	0.612	0.587	0.478	0.431
DBLP	0.834	0.834	0.794	0.790	0.621	0.591

Table 3: Mean average precision (MAP) for DBLP and IMDB queries

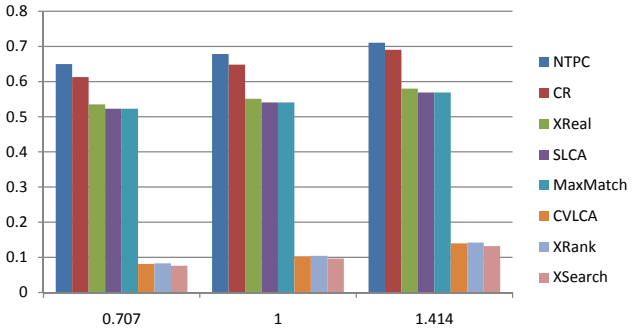

Figure 10: Average F-measure for IMDB queries

date answer. It then extends the candidate answer to be a root-subtree, by adding the path from the root of that answer to the root of the DB. SA3 looks up the NTPC of the root-subtree pattern of the candidate answer, and ranks the answer based on its NTPC. We use a modified version of the query processing algorithm from [23] (call it SA2), which is in turn an extension of the SA algorithm from [8]; we refer the reader to those papers for details and a performance analysis. In the remainder of this section, we focus on SA3's unique features.

SA and SA2 use an inverted index to find nodes that contain the query terms. SA does not present answer values to the user; SA2 finds them by a sequential scan of the original XML files, which is slow. SA3 incorporates a new storage strategy and indexes to eliminate this problem.

At startup, SA3 reads the XML data set in a depth first fashion and stores it in a NODES table in Berkeley DB [2]. Each node is described in a separate tuple of NODES. The key of each tuple is the node's Dewey code and the remainder of the tuple contains its tag name, its parent's Dewey code, and its value if it is a content node.

SA3 builds an inverted index for the text information in the NODES table. Each entry in the inverted index for a term w contains two separate lists. The first list describes the leaf nodes whose values contain w. Each list entry contains the Dewey code of the node and the number of times w occurs in its value. The second list stores the number of times w occurs in the values of each node type (all the nodes with the same tag name constitute a node type). This information is required to compute the IR based ranks, as explained below. We store the inverted index as a table MATCHES in Berkeley DB.

SA2 returns only the answer subtrees. However, for many queries the user wants to also see the siblings of the leaf nodes, which can satisfy the user's underlying information need. For instance, for the query *XML Integration* in Fig. 1, returning the subtree rooted at node 5 is not useful. The user already knows the title, and probably wants to learn the author and/or booktitle of the paper. To address this need, SA3 presents the full subtree to the user. To this end, SA3 builds an additional index CHILDREN on the parental information of the DB nodes, and stores it in Berkeley DB. Given the Dewey ID of a node, a lookup in CHILDREN finds its children. For each candidate LCA, SA3 queries CHILDREN to reconstruct the full subtree.

6. EVALUATION

We evaluated ComputeNTPC and SA3 on a Linux machine with 2GB memory and a 2.13 GHz Intel Xeon CPU. We used two real-world and one synthetic data set: DBLP (529.9 MB, max depth 5), IMDB (994.8 MB, max depth 7), and XMark (1172.3 MB, max depth 11) [20]. We use XMark only to evaluate the scalability of ComputeNTPC, as there is no way to evaluate ranking quality for queries over a nonsense database. IMDB includes long text fields such as *plot*, *quote*, *crazy-credit* (interesting facts about movies' credits), *tagline*, *goofs* (mistakes in the movies), *trivia*, and others; these fields have not been included in any previous study of the effectiveness of XML keyword query processing strategies (including [23]), and serve to illustrate the effectiveness of NTPC in the presence of long text fields. DBLP does not have long text fields; we use DBLP to show that NTPC also works well when the DB does not have very long fields, and to demonstrate ComputeNTPC's scalability.

We set $MCAS$ to 5 for DBLP and IMDB, which is a generous setting for both domains.

6.1 NTPC Computation Efficiency

Table 1 shows the time to load data, build supporting indexes, and compute NTPC, for different values of ϵ. IMDB is close to twice as large as DBLP and is more nested, so it has more patterns and more CI indexes than DBLP, and IMDB's CIs are larger on average than DBLP's. IMDB also has longer text fields than DBLP, so its join tables are larger. Thus, with $\epsilon = 0$, ComputeNTPC takes a bit more than twice as long for IMDB as for DBLP. The average length of XMark fields is less than for IMDB, but XMark is about 20% larger and is more nested, so ComputeNTPC takes about 25% longer for XMark than for IMDB, with $\epsilon = 0$.

The NTPC for a pattern is a characteristic of the underlying domain. Once NTPCs are computed for a representative instance, they do not need to be recomputed until a structural update introduces new schema element types. Thus NTPC calculations will be rare for a populated DB, and the .5-2.5 day running times shown in Table 1 are reasonable for a rare task.

Larger values for ϵ reduce the preprocessing time considerably. Run times drop off sharply at first as ϵ increases, then begin to level off, for a total drop of roughly 1/3 before ϵ grows too large. DBLP has more relatively infrequent terms than IMDB does. Thus, pruning infrequent terms reduces DBLP processing time the most. XMark content is randomly selected from a specific text collection, so XMark has relatively few infrequent terms and relatively many very frequent terms, compared to DBLP and IMDB. Thus, removing the terms with relative frequency greater than $1 - \epsilon$

	NTPC	CR	XReal	SLCA	MaxMatch	CVLCA	XSearch	XRank
Precision	0.611	0.599	0.566	0.566	0.545	0.048	0.046	0.050
Recall	0.985	0.965	0.918	0.798	0.798	0.975	0.976	0.975

Table 2: Average precision and recall for IMDB queries

helps to reduce the processing time for XMark. If ϵ grows too large, highly correlated terms can be pruned, which will change NTPCs enough to change the ranking of query answers. For DBLP, this happens for $\epsilon > .02$. Since IMDB has longer fields than DBLP, this happens at $\epsilon > .03$ for IMDB.

The final table of NTPCs was under 2MB for both DBs. Thus NTPCs can reside in main memory at query time.

6.2 Ranking Effectiveness

We used the IMDB and DBLP query workload from [24] to evaluate NTPC ranking quality. The queries came from 15 users who did not participate in the research. Each user submitted up to 5 queries to IMDB and DBLP, resulting in 40 queries for IMDB and 25 for DBLP. The exact workload can be found in [24]; we cannot list it here due to space limits. Users were given query results through a GUI interface where they can score each result as being relevant or irrelevant. We used the NTPCs computed for $MCAS = 5$ and $\epsilon = 0.02$ for DBLP, and $\epsilon = 0.03$ for IMDB. No candidate answer had more than 4 leaf nodes, which shows that $MCAS = 5$ was reasonable.

We first compared precision, recall, and F-measure [17] of NTPC against current methods: SLCA [27], MaxMatch [15], XRank [7], CVLCA [13], XSearch [5], XReal [1], and CR [23]. *Recall* gives the fraction of the relevant candidate answers that are included in the actual answer returned to the user. *Precision* gives the fraction of the returned answers that are relevant. The *F-measure* shows the tradeoff between precision and recall; it is computed as:

$$F = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}. \quad (5)$$

Setting $\beta = 1$ weights precision and recall equally. Values of $\beta < 1$ emphasize precision, while $\beta > 1$ emphasizes recall.

For DBLP, NTPC produced the exact same ranking for every query as did CR. In turn, CR has previously been shown to equal or outperform each of the six other methods on the same workload and DB instance, in terms of precision, mean average precision (for approaches that rank their answers), recall, and F-measure [23, 24]. This shows that NTPC maintains CR's good performance when the data set contains short text fields. DBLP text fields are relatively short, and none of the 25 queries in the DBLP workload demonstrated NTPC's potential advantage over CR when fields have similar text, such as "EDBT 2009" and "EDBT 2010". Since the performance of CR and the other six approaches for the DBLP instance and workload has been analyzed in previous work, we focus on IMDB in the remainder of this section.

Table 2 summarizes the recall and precision of all 8 methods on all 40 IMDB queries by averaging over all queries in the workload. NTPC had higher recall on IMDB queries than other methods including CR. The recall of the other approaches is lower, due to imperfect pruning heuristics. For example, SLCA and MaxMatch showed even lower recall on IMDB than they did for DBLP and for the data-oriented version of IMDB used in [24], which had its long text fields removed. Many relatively unimportant short and long text fields such as *crazy-credit* and *quote* are in the lower levels of the XML tree. Some important fields like *title* appear high in the tree. SLCA and MaxMatch remove answers whose roots are ancestors of other candidate answers. Thus, they omit many relevant answers. For instance, they do not return the obvious answers to *Crime The Godfather* and *High School Musical*; instead they return the plots of some crime movies, and only the plot for the "High School Musical" movie.

XReal also delivers lower recall for IMDB, compared to DBLP and the data-oriented version of IMDB. Since many TV shows, stored in *show* elements, do not have long text fields, their values are shorter. Thus they have fewer occurrences of query keywords, compared to *movie* elements. Therefore XReal filters out TV shows. However, our users wanted to see TV shows as well as movies, for example in the queries *Pearl Harbor* and *Christian Bale*. Even with long fields not a factor, XReal filters out all *show* elements for *Christian Bale*, as he appears mostly in movies. The relative precision and recall of XRank, CVLCA, and XSearch were the same, as they do not filter based on the content or the level of the root of the candidate answers.

Keyword search approaches that do not rank their answers are frustrating to use; for example, *High School Musical* returns over 100 answers under the baseline approach. We used *mean average precision* (MAP) to compare the ranking quality of all current XML keyword search approaches that do rank their answers: XRank, XSearch, XReal, CR, and NTPC. MAP shows how many of the relevant answers appear near the beginning of the returned list [17]. To compute MAP, we first consider each query Q separately. We compute the precision of all returned answers for Q, up to and including the ith relevant answer, for each value of i. The average of these precisions is called the *average precision* for Q. The MAP is the mean of the average precisions for all queries in the workload.

To compare NTPC with the content-based ranking of XSearch, XReal, and CR, we combined NTPC with pivoted normalization (PN) [17], an IR-style content ranking formula that we customized for XML. We control the relative weight of NTPC and PN as follows:

$$r(t) = \alpha NTPC(t) + (1 - \alpha) ir(t), \quad (6)$$

where $ir(t)$ is the content score of the candidate answer, computed based on the classical PN formula. α is a constant that controls the relative weight of structural and contextual information in ranking. If α is set to 1, the formula uses only structural information. If $\alpha = 0$, we have pure PN. Based on our empirical evaluation, we set the value of α to 0.8 when combining it with NTPC.

Table 3 shows the MAPs of the methods. XRank shows a relatively low MAP, as the movie domain and IMDB are not appropriate for PageRank-like heuristics. PN delivers a low MAP as well. Generally, PN ranks smaller fields and fields with more query keyword occurrences higher. There

are many fields of average length but different importance in IMDB, such as *title*, *crazy-credit*, and *tagline*, which have many words in common. For instance, for the query *Artificial Intelligence*, PN ranked some science fiction movies that have the terms *Artificial Intelligence* in their *tagline* and *plot* first, but the desired answer was the movie "Artificial Intelligence". XReal has the same problem, as it uses IR techniques. XReal assumes that fields with more occurrences of a query keyword are more important. For instance, the keywords of query *Edward Norton* appear more often in field *actor* than field *producer*. Thus, XReal ranks movies where Edward Norton acted higher than the movies he produced. Unfortunately, this heuristic works poorly for long fields, as they contain many words found elsewhere in the DB, and the unimportant (for the query) long fields may contain more of the query keywords than important fields do. For example, for query *Beautiful Mind*, XReal concludes that *tagline* is more important than *title*, because *tagline* has more occurrences of the word *beautiful* than *title* does, and ranks the desired answer with title "Beautiful Mind" low. XReal also ranks subtrees higher if they have many occurrences of the query keywords. Each movie has many long text fields in IMDB. Therefore, movies with similar subjects tend to have almost the same number of query keyword occurrences in their subtrees. For instance, all sequels to "The Mummy" have the same number of occurrences of *Mummy* and *Return*. However, the best answer to the query *Return of the Mummy* is the one with these words in its title, and XReal does not recognize this. This situation is common in data sets with long text fields.

XSearch uses an IR-style formula to rank its results. Thus, it has the same precision problems as PN. Moreover, it ranks the smaller subtrees higher. Thus, for the query *Crime The Godfather*, it ranks a movie whose keywords include *crime* and *Godfather* first, instead of the movie "The Godfather", which was the desired answer. XSearch has better MAP than PN and XRank, almost as good as that of XReal. CR performs only slightly better than PN, because CR does not work well for long text fields, as expected. NTPC correctly recognizes the important fields and patterns, and delivers better MAP than all other methods. For instance, for *Artificial Intelligence* and *Return of the Mummy*, NTPC ranks the desired movies first.

Fig 10 shows the F-measure of the 8 methods on IMDB; higher F-measures are better. NTPC's handling of long fields allows it to surpass CR, which in turn is significantly better than the other methods.

The MAP of NTPC is high but not perfect; NTPC does not return the perfect ranking for every query. The first reason is that some important fields have low collective entropy, such as *Genera* in IMDB. NTPC does not realize how important they are. The second reason is that our users preferred the most popular/famous papers and movies. IMDB and DBLP do not provide popularity information, although some related fields are present, such as *award* in IMDB. Resolving these issues is left as future work.

7. CONCLUSIONS

This paper has addressed a new challenge for XML keyword search: how to provide high-quality, well-ranked query answers when the data is highly structured but also contains long text fields such as paper abstracts or movie plot lines. Previous approaches to XML keyword search do not rank query answers well for such databases. We have proposed a approach to measure the correlation between both short and long text fields in such data sets, based on the new concept of normalized term presence correlation (NTPC). After a one-time computation of NTPCs, candidate answers for all subsequent queries are quickly ranked, based on the NTPC of their structure.

We performed a user study to evaluate the effectiveness of a NTPC-based keyword query system for data-oriented XML with and without long text fields. The study showed that the NTPC-based approach works as well as the best system previously available for data sets without long fields, and performs better than any previous system for data sets that include long fields. Since the naive method to compute NTPCs is very slow, we introduced optimization techniques and provided a faster algorithm to compute the correlation for schema elements. Our experiments on real and synthetic data sets of size .5-1.1 GB showed that NTPCs can be computed for data sets of this size in .5-2.5 days, which is reasonable for a one-time computation.

8. ACKNOWLEDGMENTS

We thank ChengXiang Zhai for helpful discussions and feedbacks. This work is supported by NSF grant number 0938071.

9. REFERENCES

[1] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective XML Keyword Search with Relevance Oriented Ranking. In *Proceedings of the Twenty Fifth International Conference on Data Engineering*, 2009.

[2] Berkeley DB. http://www.oracle.com/technology/products/berkeley-db, 2008.

[3] S. Brin, R. Motwani, and C. Silverstein. Beyond Market Baskets: Generalizing Association Rules to Correlations. In *Proceedings of the Twenty Third ACM SIGMOD International Conference on Management of Data*, 1997.

[4] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing Scoring Functions and Indexes for Proximity Search in Type-annotated Corpora. In *Proceedings of the Sixteenth International World Wide Web Conference*, 2007.

[5] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSearch: A Semantic Search Engine for XML. In *Proceedings of the Twenty Ninth International Conference on Very Large Data Bases*, 2003.

[6] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley, 1983.

[7] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of the Twenty Ninth ACM SIGMOD International Conference on Management of Data*, 2003.

[8] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE Transactions on Knowledge & Data Engineering*, 18(5):525–536, 2006.

[9] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic Discovery of

Correlations and Soft Functional Dependencies. In *Proceedings of the Thirtieth ACM SIGMOD International Conference on Management of Data*, 2004.

[10] G. Kazai and A. Doucet. Overview of the INEX 2007 Book Search track: BookSearch 07. *ACM SIGIR Forum*, 42(1), 2008.

[11] Y. Ke, J. Cheng, and W. Ng. Mining Quantitative Correlated Patterns Using an Information-Theoretic Approach. In *Proceedings of the Twelfth Annual SIGKDD International Conference on Knowledge Discovery and Data Mining* , 2006.

[12] I. Kojadinovic. Relevance Measures for Subset Variable Selection in Regression Problems Based on K-additive Mutual Information. *Computational Statistics and Data Analysis*, 49:1205–1227, 2005.

[13] G. Li, J. Feng, J. Wang, and L. Zhou. Effective Keyword Search for Valuable LCAs over XML Documents. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management*, 2007.

[14] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, 2004.

[15] Z. Liu and Y. Chen. Reasoning and Identifying Relevant Matches for XML Keyword Search. In *Proceedings of the Thirty Fourth International Conference on Very Large Data Bases*, 2008.

[16] S. Ma and J. L. Hellerstein. Mining Mutually Dependent Patterns. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, 2002.

[17] C. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2008.

[18] S. Morishita and J. Sese. Traversing Itemset Lattices with Statistical Metric Pruning. In *Proceedings of the Twelfth Annual SIGKDD International Conference on Knowledge Discovery and Data Mining* , 2006.

[19] A. Schmidt, M. Kersten, and M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *Proceedings of the Seventeenth International Conference on Data Engineering*, 2001.

[20] A. R. Schmidt et al. XMark: A Benchmark for XML Data Management. In *Proceedings of the Twenty Eighth International Conference on Very Large Data Bases*, 2002.

[21] C. Sun, C. Chan, and A. Goenka. Multiway SLCA-based keyword search in XML data. In *Proceedings of the Sixteenth International World Wide Web Conference*, 2007.

[22] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML using a Relational Database System. In *Proceedings of the Twenty Eighth ACM SIGMOD International Conference on Management of Data*, 2002.

[23] A. Termehchy and M. Winslett. Effective, Design-Independent XML Keyword Search. In *Proceedings of the Eighteenth ACM Conference on Information and Knowledge Management*, 2009.

[24] A. Termehchy and M. Winslett. Effective Ranking of XML Keyword Search Results (Extended Version), University of Illinois, UIUCDCS-R-2009-3043, 2009.

[25] S. Watanabe. Information Theoretical Analysis of Multivariate Correlation. *IBM Journal of Research and Development*, 4(1):66.

[26] J. Wen, J. Nie, and H. Zhang. Clustering User Queries of a Search Engine. In *Proceedings of the Tenth International World Wide Web Conference*, 2001.

[27] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *Proceedings of the Thirty First ACM SIGMOD International Conference on Management of Data*, 2005.

[28] M. Zaki. Efficiently Mining Frequent Trees in a Forest. *IEEE Transactions on Knowledge & Data Engineering*, 17(8):1021–1035, 2005.