A Co-Training Framework for Searching XML Documents

Wilfred Ng and Lau Ho Lam

Department of Computer Science

The Hong Kong University of Science and Technology

Hong Kong

Email: {wilfred, lauhl}@cs.ust.hk

Abstract

In this paper, we study the use of XML tagged keywords (or simply key-tags) to search an XML fragment in a collection of XML documents. We present techniques that are able to employ users' evaluations as feedback and then to generate an adaptive ranked list of XML fragments as the search results. First, we extend the vector space model as a basis to search XML fragments. The model examines the relevance between the imposed key-tags and identified fragments in XML documents, and determines the ranked result as an output. Second, in order to deal with the diversified nature of XML documents, we present four XML Rankers (XRs), which have different strengths in terms of similarity, granularity, and ranking features. The XRs are specially tailored to diversified XML documents. We then evaluate the XML search effectiveness and quality for each tailored XR and propose a Meta-XML Ranker (MXR) comprising the four XRs. The MXR is trained via a machine learning training scheme, which we term the Ranking Support Vector Machine (RSVM) in a Co-training Framework (RSCF). The RSCF takes as input two sets of labelled fragments and feature vectors and then generates as output adaptive rankers in a learning process. We show empirically that, with only a small set of training XML fragments, the RSCF is able to improve after a few iterations in the learning process. Finally, we demonstrate that the RSCF-based MXR is able to bring out the strengths of the underlying XRs in order to adapt the users' perspectives on the returned search results. By using a set of key-tag queries on a variety of XML documents, we show that the precision of the result of the RSCF-based MXR is effective.

1 Introduction

As the amount and use of XML data continue to grow, simple but effective XML searching facilities are important for users to find their target information. XML searching is different from HTML searching in two main aspects. First, XML documents differ from HTML documents in syntactical specification. In particular, XML allows the use of tags to capture data semantics. Therefore, the conventional method of using simple keywords to search XML documents is not effective. The problem has just started to be addressed in both the database and information retrieval (IR) communities [26,11,21,13,22,1,3]. Second, the tasks in XML searching are fundamentally different from those in HTML searching. Searching HTML documents belongs to the "document paradigm", which relies on a pure information retrieval approach, such as matching the important words (or keywords) between the submitted query, which is specified in a list of keywords, and the documents, which are presented in an index database. However, searching XML documents depends on how the keywords are placed in a context specified by a simple path expression. Fragments can be generic or specific in different searched contextual paths. Furthermore, XML documents are known to be diverse by nature: they vary from regularly structured documents, such as DBLP data [30], to heavily textual-oriented documents, such as Shakespeare play data [9].

Searching for information via a search engine is crucial to the experience of both casual users and professional Web programmers. In practice, most users do not actually use complicated search strategies and the advanced search options that rely on complex query formulations are therefore simply ignored. Therefore, XQuery [38] expressions are not appropriate in XML document searching, although this type of query is known to be expressive enough to derive any fragment of a given XML document precisely. The reason for this is that XQuery fulfils different goals in XML searching. XQuery is expected to return a precise XML fragment as an answer, so it focuses on evaluation efficiency when processing a submitted query. In XML searching, we focus on returning an effective ranked list of XML fragments or documents as an answer to the query. Such searching is concerned with different dimensions of relevance as well as ranking effectiveness. Given an XML search query, the prime tasks are to devise an innovative way to estimate the relevance between the query and XML documents and to rank the possible returned fragments in the search result.

In the IR research community, there have been many searching techniques developed that essentially rely on a set of weighted keywords in a search query to determine the proximity of the query and a document in the feature space. Searching XML documents, however, departs from the conventional "information retrieval" strategy, in the sense that XML documents have nested XML elements and semantics of data values indicated by tags. As a result, the notion of keyword proximity used in IR is too simple to be effective in XML searching. On the one hand, we should preserve the simplicity of search query expressions in order to facilitate the wider use of XML search engines. On the other hand, we should take into account the importance of tag semantics, document diversity and the structural complexity in XML data along with evaluations of the returned results.

We propose to study the use of a list of keywords enclosed with tags, which we term a list of *key-tags* or *a key-tag search query*, such as "⟨journal⟩ ACM ⟨/journal⟩", to search XML documents. Our method differs from the traditional keyword search, which only uses the keywords "ACM"or "ACM journal" in a search engine. A key-tag is a simple means to provide more accurate semantics for searching XML data; in this example, the key-tag means that ACM is a journal in the search query. However, using an arbitrary combination of these two words may give rise to inaccurate interpretations when matching relevant fragments or documents.

We demonstrate the viability and the benefits of using key-tags in searching XML data by devising four ranking schemes. These schemes take into account different similarities in XML contexts and granularity and lead to the development of four corresponding XML Rankers (XRs). We evaluate the proposed XML rankers via experiments using a spectrum of real XML benchmark datasets. We show that each XML ranker has its individual strengths in attaining good precision and ranking quality in different XML documents. In order to adapt the XML rankers to a wider spectrum of XML databases and users' evaluations, we use a limited training data set to train a Meta-XML Ranker (MXR) comprising the four XRs, in a machine learning framework called the RSCF. Essentially, the RSCF algorithm requires only a small set of training data of returned results to adapt the rankers in the learning process.

RSCF is an enhancement of the Ranking Support Vector Machine algorithm (the RSVM algorithm) proposed in [7], which is a machine learning technique that is applied here to optimize the performance of XML searching via key-tags. RSCF incorporates the ranking support vector machine into the co-training framework [8] to make the learning process efficient, when the amount of training data is relatively small and sparse. RSCF analyzes the ranking results of the Meta-XML Ranker by users and categorizes the results into *labelled* and *unlabelled* data sets. The labelled data sets contain a few search items that have been classified as *relevant* and the unlabelled data set contains the data items that have not yet been classified. We then augment the labelled data with the unlabelled data and rerank the results according to their relevance to obtain a larger data set for training the rankers in RSCF.

Our main contributions related to searching XML data are as follows:

- We develop a novel Meta-XML Ranker (MXR), which evolves from four XML rankers that have different strengths to various XML document categories such as data-centric and document-centric ones. The MXR is thus able to cater to mixed searching needs when given a corpus of diversified XML datasets.
- We propose a training framework based on a co-training technique called RSCF, which refines the rankers in a progressive but non-intervening manner by learning users' feedback on the returned search results. The required samples of labelled fragments are small, which means very low cost in training. The search performance of the trained ranker is shown to be able to adapt to the users' preferences in XML searching.
- We extend the Vector Space Model (VSM) that supports searching via key-tags. The extension includes the important features in XML data such as

key-tag proximity, data granularity and contextual paths as basic indexing units. The notion of similarity in the context of XML searching that is commonly used in traditional IR theory has been extended to a range of similarity features.

- We illustrate various techniques concerning practical and effective key-tag searching to compute the ranking of XML fragments as the answer to an imposed query. We then develop four XML Rankers (XRs) that satisfy desirable features of some XML datasets. We study their effectiveness with a variety of XML datasets.

In the rest of this section, we discuss related work on XML searching and co-training techniques. In Section 2, we clarify the fundamentals of key-tags. In Section 3, we discuss the technique of matching XML fragments with key-tags and the development of the four proposed ranking schemes. We define the four ranking schemes in order to deal with the diversity of real XML datasets. In Section 4, we discuss the co-training strategies for the rankers and the development of an RSCF-based Meta-XML ranker. In Section 5, we discuss the experimental results that show the effectiveness of the RSCF and the rankers. Finally, we offer concluding remarks in Section 6.

1.1 Related Work

XRANK [21] is a recently proposed XML search engine to generate ranked results for keyword search queries of hyperlinked XML documents. The engine adopts a ranking formula based on the PageRank algorithm [33] used by an existing Web search engine. The proposed formula measures extensive hyperlinks and containment edges (referencing properties) of XML elements. As

XRANK treats tag names and data values uniformly, it is not clear how the system is able to cater to the semantics of tagged data or fragments with different granularities in the search process. In contrast, we take into account the granularities of XML tags in ranking XML fragments. The referencing properties can also be captured as one of the feature components. We also aim at obtaining quality ranking results via a novel application of the co-training technique.

We share a similar spirit with XSEarch [13] in using a simple query language that is based on key-tags for XML searching. The work presents the experimental results concerning the search quality of XSEarch. However, the datasets tested are only two XML datasets, SIGMOD and DBLP, which are typical data-centric documents. The performance of XSEarch when used to search more diverse XML datasets is not clear. Compared with XSEarch, we develope four different rankers that take into account a wide spectrum of features of XML data. However, XSEarch only adopts a few ranking features and the interconnection relationship of XML nodes. We also carry out experiments on an extensive set of XML documents which consist of both data-centric and document-centric XML benchmarks.

The very recently proposed FleXPath [1] considers XPath queries of structures as a "template" and finds the best matching between the template and the full-text search. FleXPath provides ranging schemes for "top-K queries" that are interpreted in a formalized notion of relaxation semantics. The semantics essentially approximate a given query expression and retrieve the answers for a more general class of queries in the sense of query containment. Admittedly, the relaxation queries are more expressive than our key-tag search queries. However, our approach is not comparable to this work, since we do not aim to develop a soft interpretation of an existing class of precise queries such as XPath or XQuery. Another essential difference between FleXPath (or XSEarch) and our method is that we propose and study the use of the RSCF technique, which is an effective mechanism to bring out appropriate strengths of individual rankers with respect to users' searching preferences.

Support Vector Machine (SVM) [7] falls into the category of *supervised learning algorithms* in machine learning theory, which have been applied to improve search engines. The basic principle is as follows. First, SVM receives a set of training data in which each item is marked with a class label. Then, by using the labelled data, the learning algorithm generates an effective *classifier*. Co-training [8] is a new semi-supervised learning technique proposed by Blum and Mitchell. This technique provides a framework to augment the labelled data with unlabelled data and then the learning algorithm is run on the augmented training data set. Analyzing labelled data is a useful means to improve the ranking, since this method conveys *partial* relative relevance judgments on the fragments. Joachims proposed a ranking SVM algorithm that uses click-through data to optimize the performance of a retrieval function in search engines [28]. The limitation of Joachims' algorithm is that it requires a large set of training data to make the algorithm effective. In contrast, our work takes only a very small set of labelled XML fragments to train a combination of rankers in a progressive manner.

2 XML Key-Tag Queries

In this section, we discuss how to search XML fragments via a list of tagged keywords (or simply key-tags).

In defining the search queries, we maintain the spirit of using a simple combination of key-tags, since complicated search functions are largely ignored by people in reality. We now formalize the ideas related to key-tag queries.

Definition 2.1 (Key-Tag and Key-Tag Search Query) *Let* Π *be the set of tags or element names and* Σ *be the set of tagged data values (i.e., PCDATA) in an XML database. Let* $t \in \Pi \cup \{*\}$ *and* $w \in \Sigma \cup \{*\}$*. We define a* key-tag, $k = (t, w)$*, which can be viewed as a usual tagged form of an XML element "*$k = \langle t \rangle\ w\ \langle /t \rangle$*". A* key-tag search query*, denoted as* Q *(or simply a search query whenever no ambiguity arises), is a sequence of non-repeated key-tags.*

The semantics of a search query is that a fragment, F, is considered as a result candidate if at least one key-tag, k, is found in the XML fragment. In this case we say that F contains k or k is contained in F. An XML fragment can be regarded as a subtree of a given XML document that is viewed as a DOM tree. If there is more than one fragment containing the same key-tag, we only consider the fragment with the longest path from the root to the matched key-tag as a result candidate. The following definition describes the key-tag query semantics, which can be regarded as a special case of the relaxation query semantics that were recently proposed in [1].

Definition 2.2 (Search Query Semantics) *Let* K *be a non-empty subset of key-tags that are listed in the query,* Q*. We define an XML fragment,* F*, in a given document, to be a result candidate in the answer with respect to* Q*, if there exists some* K *such that all key-tags in* K *are contained in* F*. Let* F_1 *and* F_2 *be two fragments containing* K*. If* F_1 *is a subtree of* F_2*, we allow* F_1

to be the only result candidate.

For example, the search query in Figure 1 aims at finding the XML data of papers entitled “XML” written by “Mary” in year “2003” Note that the ordering of key-tags conveys a top-down view of searching. However, the query is still valid, even when the order does not match with the hierarchy of the searched XML documents. We take into account the matching between the key-tag order in a given query and the usual order in an XML data tree such as the parent-child order or the sibling order, which will be detailed in the defining feature selection. We will also show later that different ranking schemes make use of the document hierarchy to generate different ranking results.

$$\boxed{Q = \begin{array}{l} (\langle\text{author}\rangle\ \text{Mary}\ \langle/\text{author}\rangle, \\ \langle\text{title}\rangle\ \text{XML}\ \langle/\text{title}\rangle \\ \langle\text{year}\rangle\ 2003\ \langle/\text{year}\rangle) \end{array}}$$

Figure 1. An example of a key-tag search query

The main advantage of using a key-tag query is that the simple query expression relieves much burden on the majority of users, who might have been used to usual searching via keywords, to formulate more complex query expressions such as in XPath and XQuery. Compared with standard XML query languages such as XQuery, the set of key-tags is an imprecise query expression, rather than an expressive query expression that is formulated to obtain a precise XML fragment. Relatively speaking, the expressiveness of tagged search query expressions is very limited. However, the objective of this work for studying key-tag search queries is to provide an intuitive and convenient syntax to specify the search requirement.

Note that the key-tag search query offers much flexibility in terms of representation and users' search needs. We are able to provide a simple interface for entering key-tag words, which is not very different from the usual practice of entering simple keywords in existing search engines. For example, we are able to design a simple form-based interface as shown in Figure 2(a) for entering both tag names and textual data information. In this case, we call it a *complete* key-tag. Some tags can be left empty as shown in Figure 2(b). In this case, we call it an *incomplete* key-tag, which reduces to the usual case of a usual HTML search using the keywords "Mary", "XML" and "2003". An interesting case shown in Figure 2(c) is that we may leave the word column empty; then, only the element names in the XML database will be matched with the query. Finally, we could also have the wildcard symbol "$*$" in both tag and word components as shown in Figure 2(d), which captures both "raw" text and required matching tags.

Tag	Word
author	Mary
title	XML
year	2003

(a)

Tag	Word
*	Mary
*	XML
*	2003

(b)

Tag	Word
author	*
title	*
year	*

(c)

Tag	Word
*	Mary
*	XML
year	*

(d)

Figure 2. Key-tag search queries in a form-based interface

2.2 Comparison with XQuery Full-Text Searching

The essential difference between RSCF MXR and the recently proposed XQuery full-text search [1,31,2] is twofold. First, we aim to support simple search (key-tag) queries and to provide an intuitive and convenient syntax for searching. The ranking mechanism in our work is important but hidden from the users. Second, the MXR ranker is intelligent enough to be adaptive, in the sense that the ranker is able to learn the user preferences in a non-intervening manner.

Comparatively, the extension of XQuery full-text search provides a sophisticated language syntax in XQuery expressions in order to support very fine searches. However, it is still a big challenge to determine efficient query optimization and evaluation techniques to deal with the interactions between exact querying (via the core XQuery declarative constructs) and inexact searching (via the FullMatch primitives).

At the time of writing this paper, none of the XQuery engines supported the full text expression and the score clause, including the recently developed MonetDB [16]. With TeXQuery and GalaTex [2,1], the authors introduce the precursor of the full-text search extension to XQuery. TeXQuery contains functions to express queries including phrase matching, order specifications, paragraph scope, stemming and the full-text operations used by the Information Retrieval (IR) community, such as distance predicates, synonyms, and thesauri. As mentioned in the official W3C document [39], the extension of full text searching primitives in XQuery is still in a preliminary state and there are many open issues and controversial areas that may be subject to change.

For easy comparison, we list the differences between our RSCF-based key-tag searching and full text searching primitives in XQuery in Figure 3.

3 Matching Search Queries in XML Documents

In this section, we adapt the commonly used *Vector Space Model* (VSM) [4] in the context of XML documents and key-tag queries. The VSM measures the relevance between a key-tag and a target XML fragment and provides the

RSCF-Based Key-Tag Searching	XQuery Full Text Searching
The main focus is on user-centered searching: RSCF MXR emphasizes simple search queries. User preference is a prime consideration in our study of the MXR. The ranker can learn from and adapt to users.	The main focus is on language extension: XQuery FT emphasizes developing a seamless integration with XQuery semantics. Language conformance and query optimization are the main issues for developing the FT search primitives.
A spectrum of sophisticated ranking schemes are developed and defined. The meta-ranker is adaptive to users and does not require users' interferences in the ranking.	User-intervention is needed in defining the ranking schemes. The score clause is not specified and thus the user needs to define the score computation.
The search is based on a list of simple key-tags. The search result may not be effective at the very beginning. However, it relies on SVM features to capture the searching primitives. We have four pre-defined rankers that represent very comprehensive needs.	A rich set of searching primitives is defined in the W3C draft. The system relies on the users who formulate the search expressions. Users can formulate very sophisticated and fine searching expressions according to their application needs.
The aim is not to integrate directly with XML declarative query languages. RSCF MXR needs different phases of process searching and usual declarative XML querying. However, RSCF MXR can be straightforwardly extended to capture some primitives such as word weights.	Query optimization and evaluation techniques on the proposed extension are needed. The interaction between XQuery and the Fullmatch data model is still under study. The implementation details for XQuery text searching facilities are not clear at the time of writing.
Experimental details about the effectiveness of the proposed rankers are provided.	There are no implementation or evaluation prototypes.
The language has limited expressiveness but is as simple as using a Web search engine. It is independent of any declarative XML queries.	The language is expressive but complex to learn and use. The ultimate goal is to develop the search primitives as part of XQuery syntax.

Figure 3. A Comparison of RSCF MXR and Query Text Searching

basis for XML searching. As XML documents are diversified in nature, we propose a set of XML features from which four ranking schemes are defined.

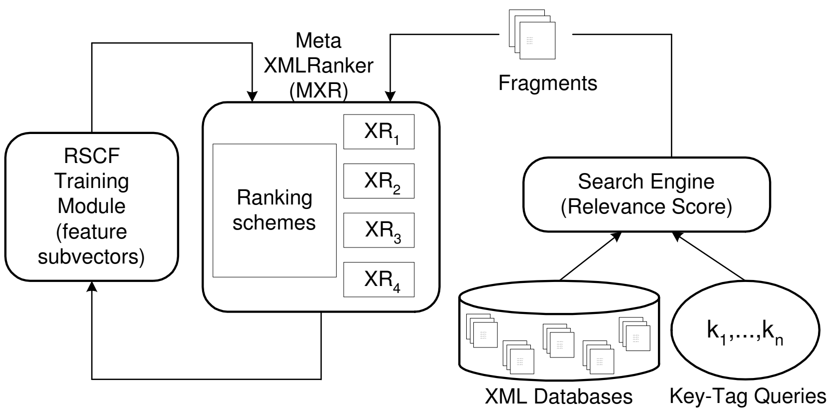


Figure 4. Overview of searching XML documents by key-tags

Figure 4 shows the basic ideas of how we search XML documents by using key-tags with the RSCF-based ranker. When the user submits a query, the search engine will search the databases for query results. We do not describe the implementation details of the XML databases and the associated searching process but remark that the query tools provided by the database vendor, such as the XML SQL Utility in Oracle [37], serve to build our prototype. We assume that the low-level search operations are efficient and thus the searching mechanism is not in the scope of our work. For each fragment in the returned result, we measure the relevance scores and pass them to the four XML Rankers (XRs). The result sets returned by the four XRs are then passed to the Meta-XML Ranker (MXR). The MXR collects user preferences and uses the RSCF algorithm to optimize the ranking functions. We can reiterate the training many times in the searching process until the MXR adapts to the user's search preferences. The effectiveness of RSCF training will be further studied in Section 5.

We first describe the indexing scheme of contextual paths in an XML corpus and then introduce the concept of an XML fragment. Intuitively, a contextual path is a sequence of tags that represents a navigation through the tree structure of the fragment starting from the root r. A contextual path expression of length n is expressed as "$r/t_1/t_2/\cdots/t_n$". This path expression specifies finding a tag, t_1, anywhere in the document, and nested in it finding a tag t_2, and so on until we find a tag t_n. Basically, a fragment, F, in the corpus can be regarded as a subtree of an XML document labelled by a contextual path in the corpus. We adopt a simple indexing scheme that numerically encodes the tag name and its occurrence in a depth-first search order of the corresponding fragment tree.

Definition 3.1 (Indexed Contextual Path and XML Fragment) *Let t be the tag name in an XML document tree, D, rooted at r. Let a_t and n_t be the corresponding numeric encoding of t and the order of occurrence of a tag in D. An indexed tag is denoted as $a_t.n_t$. Let "$p = r/t_1/t_2/\cdots/t_k$" be a contextual path consisting of k tags (r may be ignored if it is understood in the context). We define the indexed contextual path by $\rho =$ "$/a_{t_1}.n_{t_1}/a_{t_2}.n_{t_2}/\cdots/a_{n_k}.n_{t_k}$" to encode an occurrence of p in D. An XML fragment, F, specified by ρ, is the subtree of D rooted at the corresponding t_k node. We may also say that F is labelled by p, since a given indexed contextual path, ρ, corresponds to the occurrence of only one path, p.*

Following from Definition 3.1, an indexed contextual path can be as specific as a leaf element, t_l, using "$\rho = /a_{t_1}.n_{t_1}/a_{t_2}.n_{t_2}/\cdots/a_{n_k}.n_{t_l}$". In practice, a_t is a system-assigned identity of t and n_t is the same as the depth-first search order of the corresponding target node in the document tree. For example, in

Figure 5 the tags "dblp", "www" and "author" are assigned with the encodings 1, 21, and 7, respectively. The indexed tag path, $\rho =$ "/1.1/21.14/7.14", encodes the path, $p =$ "/dblp/www/author", where the order of occurrence of the tags "dblp", "www", "author" in the document fragment indexed by ρ are 1, 14, and 14, respectively.

We now introduce the algorithm that extracts and stores the index information of a collection of XML documents. The index information is stored in the following three relational tables:

(1) `documentTable (doc_ID, URL, no_of_element, max_depth, max_child)`, where `doc_ID` is a system-generated ID (being incremented by one each time) for each parsed XML document, `URL` is the URL of the XML document, `no_of_element` is the number of the XML element of this document, `max_depth` is the length of the deepest path, and `max_child` is the maximum number of child elements.

(2) `tagTable (tag_ID, tag_name, cur_ins)`, where `tag_ID` is a system assigned number encoding for the tag, `tag_name` is the label of the tag, and `cur_ins` is the count of occurrence of the tag.

(3) `keyTable (indexed_tag_path, keyword, inLink, outLink, Rank)`, where `indexed_tag_path` is the path for this keyword in the form of indexed tag path, `keyword` is the PCDATA of the element, `inLink` is the incoming references, including IDREF and XLink, `outLink` is the outgoing reference, and `rank` is the rank of the words defined by the user.

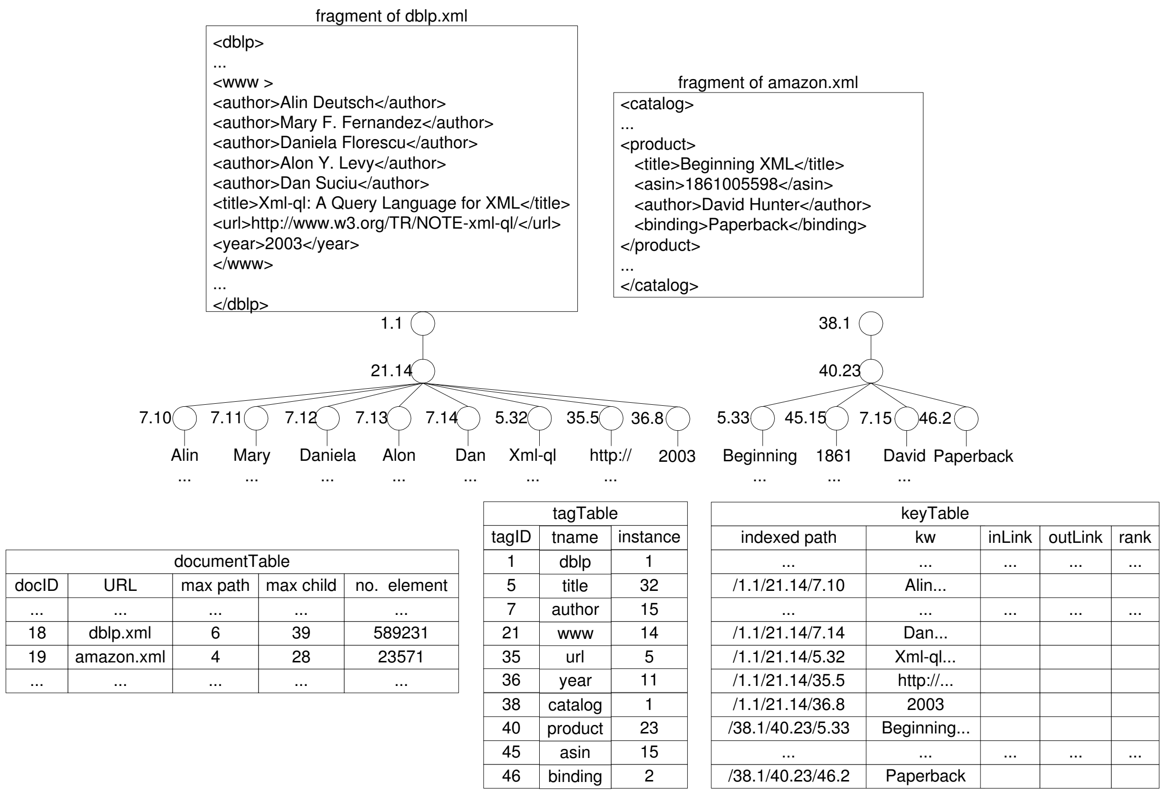
The underlying idea of Algorithm 1 is that, when an XML document is loaded and parsed, a row is inserted into the `documentTable` to store the informa-

Algorithm 1. Extracting the Contextual Path Index Information

```
Input: $T_d$: documentTable
       $T_t$: tagTable
       $T_k$: keyTable
       $D$: an input XML document
       $L$: a list variable for holding path data
Procedure:
  1: Parse the XML document starting from the root in a depth-first search
     manner;
  2: Assign $D$ with a new doc_ID = ($max$($T_d$.doc_ID) + 1)
      Case open tag or attribute, $t$, in $D$:
        if $t$ is a component in tuple $u$ of $T_t$
          Concatenate ($u$.tag_ID).($u$.cur_ins) to $L$;
          Increment $u$.cur_ins in $T_t$;
        else
          Concatenate ($max$(tag_ID)+1).(1.0) to $L$;
          Insert ($max$(tag_ID)+1) and other attributes into $T_t$;
      Case PCDATA, $w$, in $D$:
        Insert $w$ and $L$ and other attributes into $T_k$;
        Increment cur_ins in $T_t$;
      Case end tag, $t$, in $D$:
        Remove ($t$.tag_ID).($t$.cur_ins-1) from $L$;

  3: Store the information of $D$ in $T_d$
 Output: $T_d$, $T_t$ and $T_k$.
```

tion. The SAX parser then extracts the tags and converts the path into a corresponding indexed tag path based on the `tagTable`. When a tag, t, is encountered in the parser, we perform a search in the `tagTable`. If it is found with numeric encoding, a_t, and current_instance, n_t, we assign an indexed tag code, $a_t.n_t$, to the tag. If it is not found, we insert it into the `tagTable` and assign an indexed tag code, $a_t.0$, to the tag. Whenever the parser meets a text value, we store its indexed tag path, the text and link information in the `keyTable`. During the parsing, information, such as the number of elements or the maximum path depth, is collected and stored in the `documentTable`. Figure 5 presents an example that shows the tables, T_d, T_t, and T_k, immediately after two XML fragments have been parsed by the system using Algorithm 1.



fragment of dblp.xml

```
<dblp>
...
<www >
<author>Alin Deutsch</author>
<author>Mary F. Fernandez</author>
<author>Daniela Florescu</author>
<author>Alon Y. Levy</author>
<author>Dan Suciu</author>
<title>Xml-ql: A Query Language for XML</title>
<url>http://www.w3.org/TR/NOTE-xml-ql/</url>
<year>2003</year>
</www>
...
</dblp>
```

fragment of amazon.xml

```
<catalog>
...
<product>
  <title>Beginning XML</title>
  <asin>1861005598</asin>
  <author>David Hunter</author>
  <binding>Paperback</binding>
</product>
...
</catalog>
```

documentTable

docID	URL	max path	max child	no. element
...	...	...	...	...
18	dblp.xml	6	39	589231
19	amazon.xml	4	28	23571
...	...	...	...	...

tagTable

tagID	tname	instance
1	dblp	1
5	title	32
7	author	15
21	www	14
35	url	5
36	year	11
38	catalog	1
40	product	23
45	asin	15
46	binding	2

keyTable

indexed path	kw	inLink	outLink	rank
...	...	...	...	...
/1.1/21.14/7.10	Alin...			
...	...	...	...	...
/1.1/21.14/7.14	Dan...			
/1.1/21.14/5.32	Xml-ql...			
/1.1/21.14/35.5	http://...			
/1.1/21.14/36.8	2003			
/38.1/40.23/5.33	Beginning...			
...	...	...	...	...
/38.1/40.23/46.2	Paperback			

Figure 5. The index tables corresponding to two parsed XML fragments

3.2 Relevance Measure

We now extend the Vector Space Model (VSM), which adopts key-tags as an indexing unit, in order to determine the relevance between a search query and an XML fragment. We assume that a key-tag, k, is related to a fragment, F, specified by an *indexed contextual path*, ρ. The relevance score is determined by various *index weights* given by the function, Ω, as follows, which essentially extends the well-known normalized cosine measure formula (cf. [4]) to take account of key-tags and XML data in this context.

Definition 3.2 (Relevance Score) *Let Q and F be a query and an XML fragment, where F is specified by an index contextual path, ρ and p is the corresponding contextual path. Let Ω be a weight function that maps a given key-tag, k, from Q (or F) into an indexing weight (a positive constant). The mapping takes into account the two components of $k = (t, w)$. The* relevance score *of F to Q, denoted as $Sim(Q, F)$, is evaluated by using a similarity between the weight vectors of F and Q given by the following expression.*

$$Sim(Q, F) = \frac{\Sigma_{k \in (Q \cap F)} \ \Omega(k, Q) \times \Omega(k, F) \times G(k, p)}{\mid Q \mid \times \mid F \mid}. \tag{1}$$

The weights associated with the fragment are calculated based on the product of two frequency parameters, $\Omega(k, Q)$ and $\Omega(k, F)$, which indicate the statistical importance, and one path parameter, $G(k, p)$, which indicates the importance of granularity of the fragment.

(1) The key-tag frequency, $\Omega(k_i, Q)$, represents the frequency of occurrence of a key-tag, k_i, within a query $Q = (k_1, k_2, \dots, k_n)$, and is defined as

follows:

$$\Omega(k_i, Q) = \frac{2(n-i+1)}{(n+1)n},$$

where n is the total number of key-tags in Q. Here the equation is simply a normalization of the position index, i, of the key-tag with respect to the sum of all position indexes, given by $\Sigma i = \frac{n(n+1)}{2}$. As shown in the denominator, the fraction represents our consideration that a higher order of the occurrence of k_i with respect to Q, k_i, should then have a higher weight.

(2) The fragment frequency, $\Omega(k, F)$, represents the content discrimination factor, which means that if a key-tag appears often in a fragment, then it describes well the fragment contents. However, if a key-tag appears in many fragments, then it is not useful for distinguishing a fragment. We now give the definition of the fragment frequency as follows:

$$\Omega(k, F) = \begin{cases} (N_k/N_F) \times log(N/N_C) & \text{where } k \text{ is a key-tag in } F; \\ 0 & \text{otherwise,} \end{cases}$$

where N_F is the total number of key-tags in F, N_k is the number of occurrences of k in F, N_C is the number of fragments in the collection that contain k, and N is the total number of fragments in the collection. Here the equation is analogous to the well-known *tf-idf* definition (cf. [4]) in order to represent our consideration that if a key-tag is frequent in the fragment (the first fraction) and infrequent in other fragments (the second fraction), then k_i should have a higher weight with respect to F.

(3) The degree of granularity matching of k in p, $G(k,p) = t/l_p$, where t is the number of occurrences of tag in p and l_p is the length of p in F. Here,

the equation is a simple ratio to represent the specificity of k in the path p, which will be further illustrated in Figure 8.

```
<dblp>
...
<www>
<author>Alin Deutsch</author>
<author>Mary F. Fernandez</author>
<author>Daniela Florescu</author>
<author>Alon Y. Levy</author>
<author>Dan Suciu</author>
<title>Xml-ql: A Query Language for XML</title>
<url>http://www.w3.org/TR/NOTE-xml-ql/</url>
<year>2003</year>
</www>
...
</dblp>
```

Figure 6. An example XML fragment

Consider the Query, Q, in Figure 1 and the XML fragment, F, in Figure 6. We have $\Omega((author, Mary), Q) = \frac{2(3-1+1)}{(3+1)3} = 0.5$ and $\Omega((author, Mary), F) = 1/8 \cdot log(10/1) = 0.125$, assuming 10 fragments in the collection. The path, $p =$ "$/dblp/www/author$" contains the key-tag (author, Mary). The length of p is 3 and thus $G(k,p) = (1/3) = 0.3333$. It follows that the product of the three terms, $\Omega(k,Q)$, $\Omega(k,F)$, and $G(k,p)$, in the nominator of $Sim(Q,F)$ is $(0.5 \cdot 0.125 \cdot 0.3333) = 0.02083$. Similarly, we compute the nominator for the key-tags $(title, XML)$ as $(0.3333 \cdot 0.125 \cdot 0.3333) = 0.01389$ and for $(year, 2003)$ as $(0.1667 \cdot 0.125 \cdot 0.3333) = 0.006945$. Finally, we have the following relevance score.

$$Sim(Q,F) = \frac{0.02083 + 0.01389 + 0.006945}{3 \times 8} = 0.001736.$$

It is worth pointing out that if $G(k,p) = 1$ and we ignore the tag component, $k.t$, in the weight functions, then the cosine measure formula given in Definition 3.2 becomes a simple relevance measure of searching usual flat documents. There are some considerations that affect the relevance measures in the granularity matching between a key-tag and the contextual path. Let us illustrate the point by assuming $p =$ "$/c_1/\cdots/c_m/$", where "c_m" is the leaf element with (textual) data value "a", $Q = \{k_1, \ldots, k_n\}$, and $k_i = (t_i, v_i)$ for $i \in \{1, \ldots, n\}$.

There are three cases to consider as below.

- Case 1 (key-tag matching): $\exists k_i, c_j$ such that $t_i = c_j$ and $v_i = a$.
- Case 2 (word matching): $\forall k_i, c_j$, $t_i \neq c_j$ but $\exists k_i$ such that $v_i = a$.
- Case 3 (tag matching): $\forall k_i$, $v_i \neq a$ but $\exists k_i, c_j$ such that $t_i = c_j$.

In Case 1, the path granularity leads to two further choices as follows. If we prefer to have a shorter distance between the tag name, t, and its associated keyword, w, in p, then more specific results will be returned in the high ranks. In contrast, if we prefer to have a longer distance between t and w, then more generic results will be returned in the high ranks. This motivates us to devise different ranking schemes introduced later in Section 3.4.

The ranking of fragments can be calculated by using the vector equation, $\overrightarrow{\omega} \times \phi$, where $\overrightarrow{\omega}$ is a weighting vector, which specifies the weight of different features and ϕ is the feature vector. The following are the features we extract and apply to define the feature vector mapping, $\phi(Q, F)$, in different ranking schemes, which are classified into the three categories of: ranking, similarity, and granularity features. These features are all essential to evaluating an XML fragment in the search result.

(I) Ranking Features:

Let $E \in \{DOC, DAT, DFT, CUS\}$ (DOC stands for document-centric ranking, DAT stands for data-centric ranking, DFT stands for system default ranking, and CUS stands for customized ranking) [1] and the rank parameter, $T \in \{1, 3, 5, 10\}$. We define the function *Ranking Features*, *Rank*, as follows:

$$Rank(E, T) = \begin{cases} 1 \text{ if } F \text{ is ranked top } T \text{ in } E; \\ \\ 0 \text{ otherwise.} \end{cases}$$

The combination between E and T results in a total of 16 binary ranking features. We restrict T in $\{1, 3, 5, 10\}$, since we consider that the top-10 fragments are the most important returned results (cf. the discussion on Zipf's law effect in [23]). The other numbers are obtained by repeatedly performing a simple binary division on 10. Essentially, the feature provides a boolean variable that indicates whether a ranker scheme, E, is capable of positioning

[1] These four rankers will be detailed in Section 3.4. However, we need their names here to specify the basic features and explain the feature vector.

a target fragment within various top-T ranges that satisfy the user.

(II) Similarity Features:

Let $Q.\omega$ and $F.\omega$ as the sets of words (i.e., the textual values only) appearing in a query, Q, and an XML fragment, F, respectively, where $\omega \subseteq \Sigma$. Similarly, we define $Q.\tau$ and $F.\tau$ be the sets of tags in Q and F, respectively, where $\tau \subseteq \Pi$ (recall the meaning of Σ and Π in Definition 2.1).

(1) Keyword similarity: $Sim_K(Q,F)$.

Let N be the number of *non-stop words* occurring in $F.\omega$. Here, *stop words* are those words that have no meaning from the point of view of searching, such as the definite and indefinite articles in English. Otherwise, words are *non-stop words*. We denote P_+ as the frequency of the words in $F.\omega$ belonging to $Q.\omega$ (positive samples) and P_- as the frequency of the terms in $F.\omega$ not belonging to $Q.\omega$ (negative samples). We define the *keyword similarity*, denoted as $Sim_K(Q,F)$, between the query, Q, and the retrieved fragment, F, as follows:

$$Sim_K(Q,F) = \begin{cases} logN & \text{if } \forall w_i \in F.\omega, w_i \in Q.\omega; \\ -logN & \text{if } \forall w_i \in F.\omega, w_i \notin Q.\omega; \\ \frac{1}{2}log\frac{(1-P_-)P_+}{(1-P_+)P_-} & \text{otherwise.} \end{cases}$$

The equation defined above is analogous to the formulae developed in text searching [4,18]. There are three exclusive cases in the above formula. First, *all* the keywords in the fragment are found in the query. Second, *none* of the keywords in the fragment is in the query. Thus, we need to suppress this feature by a negative log formula. Finally, for the intermediate case, we compute the feature by using the log ratio measurement, which takes the frequency of both positive and negative samples into consideration. The formula is derived from Baye's theorem for probabilistic information retrieval in [19]. The log function is introduced to dampen the effect of the increase in the involved set size.

(2) Access similarity: $Sim_A(Q, F)$.

We maintain the set of n most frequently accessed key-tags, M, in the system. Let $M.\tau$ be the set of tag names in M. We define the *access similarity*, denoted as $Sim_A(Q, F)$, between Q, $M.\tau$, and F, as follows:

$$Sim_A(Q, F) = \begin{cases} 1 \text{ if } F.\tau \text{ overlaps } (M.\tau \cap Q.\tau); \\ 0 \text{ otherwise.} \end{cases}$$

This feature provides a Boolean variable that indicates if there exists a frequently accessed key-tag in Q that can also be found in the fragment.

(3) Path similarity: $Sim_P(Q, F)$.

Let $F.\rho$ be the set of all paths running from the root to leaf nodes in F. Let $Q.\rho = \{y \in F.\rho \mid \exists x \in Q.\tau$ such that x is a tag occurring in the path $y\}$. We define the *path similarity*, denoted as $Sim_P(Q, F)$, between the query, Q, and the fragment, F, as follows:

$$Sim_P(Q, F) = \frac{|Q.\rho|}{|F.\rho|}.$$

The $Sim_P(Q, F)$ is a simple ratio of the number of paths containing some key-tags of the query in the fragment to the total number of paths in the fragments. Essentially, the feature indicates the fraction of paths in the fragment that are related to the query.

(4) Element similarity: $Sim_E(Q, F)$.

We define the *element similarity*, denoted as $Sim_E(Q, F)$, between the query, Q, and the fragment, F, as the fraction of tags or words in F that overlaps those in Q:

$$Sim_E(Q, F) = \frac{|(Q.\tau \cup Q.\omega) \cap (F.\tau \cup F.\omega)|}{|F.\tau| + |F.\omega|}.$$

The feature treats both tags and keywords uniformly as usual words and computes a simple ratio of the number of common words in the query and fragment to the total number of words in the fragment.

(5) Order similarity: $Sim_{AO}(Q, F)$ and $Sim_{SO}(Q, F)$.

The order similarity can be further divided into the ancestor order similarity, AO, and the sibling order similarity, SO. Let B_Q be the set of all possible ordered pairs of tags extracted from Q, which matches the ordering of the key-tags given in Q (recall that Q is a sequence of key-tags by Definition 2.1), and let $F.AO$ and $F.SO$ be the sets of ancestor and the sibling ordered pairs of tags extracted from F. The ordered pairs in $F.AO$ and $F.SO$ match either the ancestor order or the sibling order of the searched document. We define the *ancestor order similarity* and *sibling order similarity*, denoted as $Sim_{AO}(Q, F)$ and $Sim_{SO}(Q, F)$, between the query, Q, and the fragment, F, as the fraction of $F.AO$ and $F.SO$ that overlaps those in B_Q:

$$Sim_{AO}(Q, F) = \frac{|B_Q \cap F.AO|}{|B_Q|}, \quad Sim_{SO}(Q, F) = \frac{|B_Q \cap F.SO|}{|B_Q|}.$$

Both $Sim_{AO}(Q, F)$ and $Sim_{SO}(Q, F)$ features represent the fractions of ordered pairs of tags in the query that match with their counterparts in their fragments according to AO and SO of the fragment tree. The extreme case happens when the fragment contains the query tags that are in the same order with the query. $Sim_{AO}(Q, F)$ and $Sim_{SO}(Q, F)$ then become one. On the other hand, if the fragment contains no tags of the query or the query tag order totally mismatches with that of the fragment, then $Sim_{AO}(Q,F)$ and $Sim_{SO}(Q,F)$ then become zero.

(III) Granularity Featurefs:

Let $F.r$ be the root of F. We measure the *granularity* of a retrieved fragment, F, by the following *granularity features*:

(1) Sib : The order of occurrence of fragments whose roots are siblings of $F.r$.
(2) Chi : The order of occurrence of tags whose parent is $F.r$.
(3) Dis_+ : The distance from $F.r$ to the farthest leaf node.
(4) Dis_- : The distance from $F.r$ to the nearest leaf node.
(5) Tag : The order of occurrence of tags in $F.r$.
(6) Att : The order of occurrence of attributes of $F.r$.

The granularity measure of a feature, X, for a given fragment, F, denoted as $Grn_X(F)$, is defined as follows:

$$Grn_X(F) = \frac{X(F) - avg(X)}{avg(X)},$$

where X is one of the above granularity features and $avg(X)$ is the average value of the feature, X, in the XML document where F is embedded. The granularity feature is a simple ratio of various parameters to their average value of the fragments in the search result.

We now use the following example to illustrate some computation of the above features. Assume F to be the DBLP fragment, which is shown in Figure 6. Let $Q = (k_1, k_2)$ where $k_1 = \langle author \rangle Dan \langle /author \rangle$ and $k_2 = \langle title \rangle XML \langle /title \rangle$. The keyword similarity is $Sim_K(Q,F) = \frac{1}{2} log \frac{(1-0.55)0.25}{(1-0.25)0.55} = -0.2821$. Assume that the tags $\langle title\rangle$ and $\langle author\rangle$ are the most frequently accessed tags and that the access similarity, $Sim_A(Q,F)=1$. There are eight paths in F and six of them contain tags from Q. The path similarity is $Sim_P(Q,F)=\frac{6}{8}=0.75$. The element similarity is $Sim_E(Q,F)=\frac{6}{14}=0.4286$. $Q.\tau=\{author, title\}$ and $B_Q=\{\langle author, title\rangle\}$. $F.\tau=\{dblp, www, author, title, url, year\}$. We thus have $F.AO=\{(dblp, www), (dblp, author), (dblp, title), (dblp, url), (dblp, year), (www, author), (www, title), (www, url), (www, year)\}$ and $F.SO=\{(author, title), (author, url), (author, year), (title, url), (title, year), (url, year)\}$. The ancestor and sibling order similarities are $Sim_{AO}(Q,F)=\frac{0}{1}=0$ and $Sim_{SO}(Q,F)=\frac{1}{1}=1$.

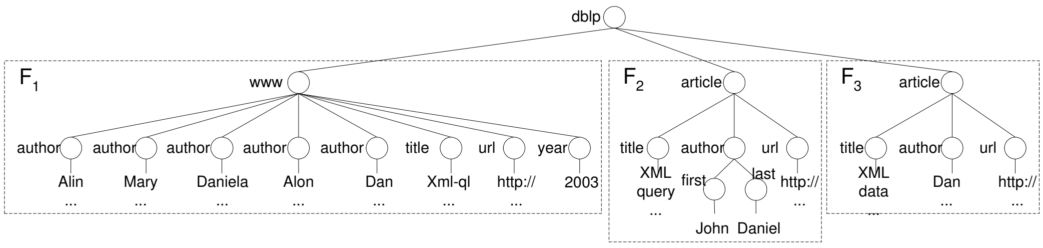


Figure 7. Three fragments, F_1, F_2 and F_3, returned by the query, Q'

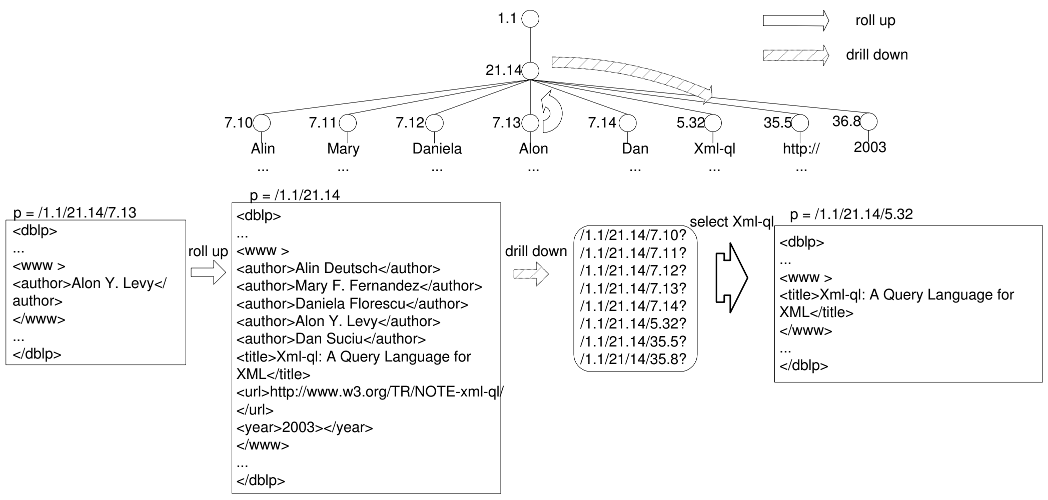
Figure 7 shows the three fragments, F_1, F_2 and F_3, which are returned by the query, $Q'=(\langle author\rangle Dan\langle/author\rangle, \langle title\rangle XML\langle/title\rangle)$. Consider F_1. Its siblings are F_2 and F_3. Therefore, $Sib(F_1)$ is 2. Similarly, $Sib(F_2)$ and $Sib(F_3)$ are also 2. We have $Grn_{Sib}(F_1)=\frac{Sib(F_1)-(Sib(F_1)+Sib(F_2)+Sib(F_3)/3}{(Sib(F_1)+Sib(F_2)+Sib(F_3))/3}=\frac{2-(2+2+2)/3}{(2+2+2)/3}=0$, which means that the number of siblings of F_1 is just average (i.e. positive Grn_{Sib} means relatively more siblings and negative means otherwise). As F_1 has 7 children, F_2 and F_3 have 3 children. We have $Grn_{Chi}(F_1)=\frac{7-(7+3+3)/3}{(7+3+3)/3}=0.6154$. In F_1, the distance from $F_1.r$ to the farthest leaf nodes is 1. Both $Dis_+(F_1)$ and $Dis_-(F_1)$ are 1. Similarly, $Dis_+(F_2)$ and $Dis_-(F_2)$ are 2 and 1, and $Dis_+(F_3)$ and $Dis_-(F_3)$ are both 1. We have $Grn_{Dis_+}(F_1)=\frac{1-(1+2+1)/3}{(1+2+1)/3}=-0.25$ and $Grn_{Dis_-}(F_1)=\frac{1-(1+1+1)/3}{(1+1+1)/3}=0$. A negative $Grn_{Dis_+}(F_1)$

means that the farthest distance from leaf nodes to $F_1.r$ is below the average distance from the root to the leaf nodes. The total number of tag occurrences in F_1, F_2, and F_3 are 9, 6, and 4. $Grn_{Tag}(F_1) = \frac{9-(9+6+4)/3}{(9+6+4)/3} = 0.4211$. The calculation for $Grn_{Att}(F_1)$ is similar to $Grn_{Tag}(F_1)$, whose value is 0, since all the fragments do not have attributes.

It is worth mentioning that we allow users to choose, further, a wider or a more specific view of the output fragments. This is due to the fact that the zooming effect can be achieved by moving the view on the indexed contextual path and then controlling the scope of a fragment in the document tree (e.g., "1.0/2.0/3.1" can be rolled up to "1.0/2.0"). Figure 8 shows an example that demonstrates the meaning of specificity and genericity (i.e., roll up and drill down views are possible). In this example, we assume that, in a search result, the fragment of the indexed contextual path, "/1.1/21.14/7.13", which is the author context, is returned. Then, the user is able to roll up the result. The parent fragment specified by the contextual path "/1.1/21.14", which is the WWW context, will be returned. The user may drill down by selecting any of the corresponding child fragments, for instance, the one specified by "/1.1/21.14/5.32", which is the title context as shown in Figure 8.

3.4 Ranking Schemes for XML Fragments

XML documents are commonly classified into two broad categories of *document-centric* and *data-centric* documents. Roughly, document-centric documents are for human consumption and contain relatively more textual data than structural data, such as books, email messages, and some XHTML documents. Document-centric documents have irregular structures, large-grained data and



```
<dblp>
...
<www >
<author>Alon Y. Levy</
author>
</www>
...
</dblp>
```

```
<dblp>
...
<www >
<author>Alin Deutsch</author>
<author>Mary F. Fernandez</author>
<author>Daniela Florescu</author>
<author>Alon Y. Levy</author>
<author>Dan Suciu</author>
<title>Xml-ql: A Query Language for
XML</title>
<url>http://www.w3.org/TR/NOTE-xml-ql/
</url>
<year>2003></year>
</www>
...
</dblp>
```

```
/1.1/21.14/7.10?
/1.1/21.14/7.11?
/1.1/21.14/7.12?
/1.1/21.14/7.13?
/1.1/21.14/7.14?
/1.1/21.14/5.32?
/1.1/21.14/35.5?
/1.1/21/14/35.8?
```

```
<dblp>
...
<www >
<title>Xml-ql: A Query Language for
XML</title>
</www>
...
</dblp>
```

Figure 8. Controlling specificity by rolling up and drilling down the contextual path

lots of mixed content. Data-centric documents are for machine consumption and XML as a data transport means is adopted. Data-centric data conform to regular structures, such as publishing relational data in XML. The order in which sibling elements occur in data-centric XML documents is generally not significant. Examples of data-centric documents are sales orders, flight schedules, scientific data, and stock quotes.

In practice, the boundary between document-centric and data-centric documents is not always clear. For example, Shakespeare [9] is commonly regarded as document-centric but DBLP is regarded as data-centric [30]. Most XML documents actually lie between these two extreme classifications. In order to optimize the ranking quality and to deal with the different categories of XML documents, we devise four ranking schemes as described below.

Document-centric ranking (DOC). In this ranking scheme, we mark a set of pre-defined tags in the tag tables (i.e., D_T). For example, in Shakespeare [9], we mark the tags of `act`, `play`, `speech`. Since this ranking scheme gives preference to document-centric data, we implement the following ranking rule: the smaller the path difference between the keywords and prefixes, the more specific the result. In addition, in document-centric documents, the depth of the path is considered to be less significant. We define the document-centric ranking, ϕ_{DOC}, as (Sim_K, Sim_A, Sim_P, Sim_E, Sim_{AO}, Sim_{SO}, Grn_{Sib}, $-Grn_{Dis_+}$).

Data-centric ranking (DAT). In this ranking scheme, we do not use a set of predefined tags, since we may not know the meaning of the tags. An example is the DNA jargon used in SwissProt [17]. In contrast to the DOC scheme, the DAT scheme gives preference to tags specified by longer paths in the data-centric document, since such paths provide more specific information. We define the data-centric ranking, ϕ_{DAT}, as (Sim_K, Sim_A, Sim_E, Grn_{Dis_+}).

System default ranking (DFT). In this ranking scheme, we consider a mix of the DOC and DAT schemes. When the documents are loaded in the system, some statistical data are collected, including the average path length and the average number of children. We rank the result that is near the highest rank as the system default. We define the system default ranking, ϕ_{DFT}, as (Sim_K, Sim_P, Sim_{AO}, Sim_{SO}, Grn_{Sib}, Grn_{Chi}, Grn_{Dis_+}, Grn_{Dis_-}, Grn_{Tag}, Grn_{Att}).

Customized ranking (CUS). In this ranking scheme, we allow users to pre-define the weighting for their query. The user can control the weighting of those factors listed in the draft. The feature, ϕ_{CUS}, and the weight vector, $\overrightarrow{w_{CUS}}$, are supplied by the users.

We implement the above four ranking schemes and develop four respective XML Rankers (XRs), each of which is able to generate its own list of search results. In order to reinforce the strength and reduce the weakness of individual rankers with respect to a collection of mixed XML data, we develop a Meta-

XML ranker (MXR) which combines the returned results of each ranker in a round-robin manner. We adopt the *Ranking Space vector machine and Co-training Framework* (RSCF) to train the MXR, which is described in Section 4. We depict the general idea of the RSCF-based MXR in Figure 9.

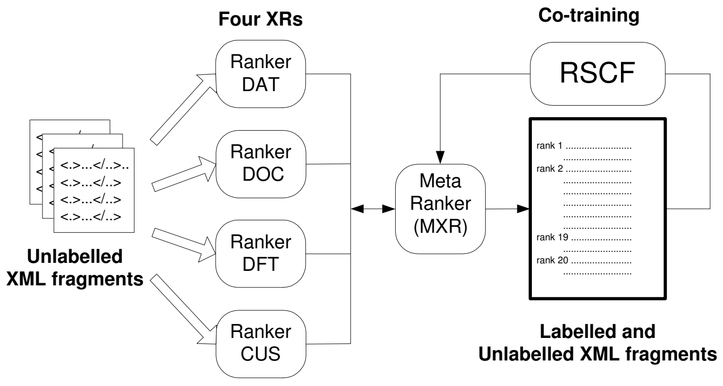


Figure 9. A conceptual view of the RSCF-based Meta-XML Ranker

4 The RSCF-Based XML Ranker

In this section, we describe a *Meta-XML Ranker* (MXR) based on the RSCF algorithm. RSCF addresses the problem that the training set of preference feedback extracted from a single query is relatively small.

4.1 Preference Fragments

We now first introduce the notion of training fragments. Given a query, Q, the returned list of ranked result is classified into two categories of labelled and non-labelled fragments. We use the set of labelled fragments as the training data. Formally, a labelled fragment is denoted as a triplet, (Q, R, C), where Q is the input search query, R is a list of ranked fragments, $(F_1, \ldots, F_n)$, and C is the set of labelled fragments that are considered to be relevant to the query. Figure 10 illustrates the submitted query, $Q = (\langle Title\rangle XML\langle /Title\rangle,$ $\langle Year\rangle 2001\langle /Year\rangle)$, and the returned list of the ranked result. In this example, we assume that a user scans the rankings from top to bottom. The three fragments, F_1, F_7, and F_{10}, are "labelled", which means that they are considered to be relevant with respect to some user preferences and thus the labelled fragments serve as the training data in the RSCF.

It is useful to make use of labelled data to improve the ranking. This is due to the fact that the data convey *partial* relative relevance judgments on the fragments. Our proposed learning function requires only a few labelled fragments to start the co-training framework. Suppose the fragments F_2, F_3, F_4, F_5, and F_6 are scanned before fragment F_7 is labelled, i.e., implying the decision that F_2 to F_6 are not so relevant. Therefore, F_7 is more relevant than the other fragments according to the sample judgment. In other words, F_7 should rank ahead of these five links in the target ranking. Similarly, F_{10} should rank ahead of F_2, F_3, F_4, F_5, F_6, F_8, and F_9. We now denote the ranking from the sample data as r' and call the ordered pair deduced from the partial relative relevance judgment *the preference fragment pair*. It is straightforward to check that the three sets of preference fragment pairs according to the three fragments, F_1, F_7, and F_{10}, can be obtained as shown in Figure 11. These three sets represent the relevance judgments collectively, where some links are incomparable (e.g., F_1, F_7 and F_{10} are incomparable with respect to $<_{r'}$).

4.2 Ranking SVM Techniques

We now discuss how to apply Ranking SVM (RSVM) to training a ranker. The RSVM first takes the set of labelled fragment pairs as input and then returns

Fragments ID	XML fragments in the search results
F_1 (labelled)	<dblp><www> ..... <title>XML Query Use Cases< /title> <year>2001</year></www></dblp>
F_2	<play><fm> ..... <p>XML version by Jon Bosak, 1996-1999.</p> <p>The XML markup in this version is Copyright</p></fm></play>
F_3	<datasets><dataset><year> ..... 2001 </year></dataset></datasets>
F_4	<datasets><dataset> ..... <year>2001</year></dataset></datasets>
F_5	<datasets><dataset><identifier> ..... *I_5.xml* </identifier></dataset></datasets>
F_6	<datasets><dataset><identifier> ..... *I_98A.xml* </identifier></dataset></datasets>
F_7 (labelled)	<dblp><www> ..... <title> XQuery: A Query Language for XML < /title> <year> 2001 </year> </www> </dblp>
F_8	<dblp><article> ..... <year> 2001 </year> </article> </dblp>
F_9	<root><entry><ref><cite> ..... Submitted (JAN-1997) to the EMBL/ GenBank/DDBJ databases </cite></ref></entry></root>
F_{10} (labelled)	<dblp><www> ..... <url><www key="www/org/w3/TR/NOTE-xml-ql">< /url> <year> 2001 </year> </www> </dblp>

Figure 10. The labelled and unlabelled data for the query Q

Set of preference fragment pairs arising from F_1	Set of preference fragment pairs arising from F_7	Set of preference fragment pairs arising from F_{10}
Empty Set	$F_7 <_{r'} F_2$	$F_{10} <_{r'} F_2$
	$F_7 <_{r'} F_3$	$F_{10} <_{r'} F_3$
	$F_7 <_{r'} F_4$	$F_{10} <_{r'} F_4$
	$F_7 <_{r'} F_5$	$F_{10} <_{r'} F_5$
	$F_7 <_{r'} F_6$	$F_{10} <_{r'} F_6$
		$F_{10} <_{r'} F_8$
		$F_{10} <_{r'} F_9$

Figure 11. Sets of preference fragment pairs derived from the labelled data

a trained ranker. The RSVM algorithm needs to tolerate some ranking errors in the training process.

Suppose r^* is the target ranking in the search result of Q. Although r^* is optimal with respect to the documents, it is *not* fully observable in practice. However, we are able to obtain r' from the labelled data, which is, in fact, a subset of r^*. Given the training set, $\{(Q_1, r'_1), (Q_2, r'_2), \ldots, (Q_n, r'_n)\}$, we aim to find a rank that holds as many preference feedback fragment pairs in r' as possible.

The principle of achieving an optimal ranking with respect to a given training set is as follows. First, by extracting a feature vector, we can rank the documents in the search result by giving different weights to the features. Then, we find a weight vector, $\overrightarrow{\omega}$, that makes the set of inequalities given in (2) hold for $1 \leq k \leq n$:

$$\forall (F_i, F_j) \in r'_k : \overrightarrow{\omega}\phi(Q_k, F_i) > \overrightarrow{\omega}\phi(Q_k, F_j). \tag{2}$$

Here, $(F_i, F_j) \in r'_k$ is a fragment pair that corresponds to the preference pair, $(F_i <_{r'_k} F_j)$, with respect to the submitted query, Q_k; $\phi(Q_k, F_i)$ is a mapping that maps Q_k onto a sequence of features (or a *feature vector*) that describes the match between Q_k and F_i. Figure 12 illustrates how the weight vector, $\overrightarrow{\omega}$, determines the ordering of the three fragments, F_1, F_2, and F_3, in two dimensions. The documents are ordered as (F_1, F_2, F_3) according to $\overrightarrow{\omega_1}$ and as (F_2, F_1, F_3) according to $\overrightarrow{\omega_2}$. The former is better than the latter if the target ranking is $F_1 <_{r^*} F_2 <_{r^*} F_3$.

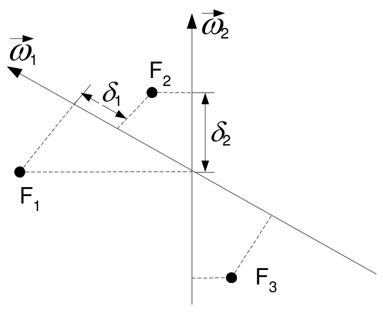


Figure 12. Ranking F_1, F_2, and F_3 according to the weight vectors, $\overrightarrow{\omega_1}$ and $\overrightarrow{\omega_2}$

The problem of solving $\overrightarrow{\omega}$ using the set of inequalities given in (2) is *NP-hard*. However, an approximated solution can be obtained by introducing a non-negative *slack variable*, ξ_{ijk}, to tolerate some ranking errors [7]. Recall that r'_k is a subset of the target ranking, r^*_k, for the search result of the query, q_k. Algorithm 2 outlines the RSVM algorithm based on the approximation. The basic idea is that if we consider that δ is the distance between the two closest projected fragments, then the larger the value of δ, the more definite the ranking, and hence the better the quality of the training result (see Figure 12). Thus, if there are several weight vectors that are able to make all the rankings subject to the condition mentioned in the RSVM algorithm, we will choose the one that can maximize margin δ. Minimizing $\frac{1}{2}\overrightarrow{\omega} \cdot \overrightarrow{\omega}$ in Algorithm 2 can be viewed as maximizing margin δ. In addition, minimizing $\Sigma\xi_{ijk}$ can be viewed as minimizing the ranking errors. Parameter C is introduced here to allow for a trade-off between the margin size and the training errors in Algorithm 2.

Algorithm 2. RSVM Algorithm

```
Input: A ranked list $r'_k$ ($1 \leq k \leq n$) extracted from the set of labelled fragment pairs;
Procedure:
Minimize: $V(\overrightarrow{\omega}, \xi) = \frac{1}{2}\overrightarrow{\omega} \cdot \overrightarrow{\omega} + C\Sigma\xi_{ijk}$;
Subject to: for all $i, j$, and $k$,
    $\forall(F_i, F_j) \in r'_k : \overrightarrow{\omega}\phi(Q_k, F_i) > \overrightarrow{\omega}\phi(Q_k, F_j) + 1 - \xi_{ijk}$;
    $\xi_{ijk} \geq 0$;
Output: $\overrightarrow{\omega}$.
```

4.3 A Co-training Framework for Effective Rankers

We now analyze the labelled data set through the RSCF algorithm and apply a co-training framework to optimize the ranking functions. The key idea of co-training is to augment the set of labelled fragments with the set of unlabelled fragments when the labelled training fragments are limited and sparse. As the training data set is enlarged, the classification errors are significantly decreased.

In the RSCF algorithm, we enhance the RSVM algorithm and incorporate the co-training framework. First, the feature vector, $\phi(Q, F)$, discussed in Section 3.3 is first divided into two feature subvectors, $\phi_A(Q, F)$ and $\phi_B(Q, F)$. Then, the two rankers, α_A and α_B, are incrementally built over these two feature subvectors. Both rankers use the RSVM algorithm to learn the weight vectors. Each ranker is initialized with a few labelled preference fragment pairs extracted from the sample data (e.g., (F_7, F_3) in Figure 10). In each iteration of co-training, each ranker chooses several preference fragment pairs (e.g., (F_9, F_8) in Figure 10) from the unlabelled data set and adds them to the labelled data set. The chosen fragment pairs are those with the highest ranking confidence as given by the underlying rankers. Then, each ranker is rebuilt from the augmented labelled set. In the next iteration, the new rankers are used to rank the unlabelled preference fragment pairs again. The ranking preference fragment pairs process and the building rankers repeat until all unlabelled preference fragment pairs are labelled or a terminating criterion is satisfied. Figure 13 illustrates the process and the underlying idea of the algorithm.

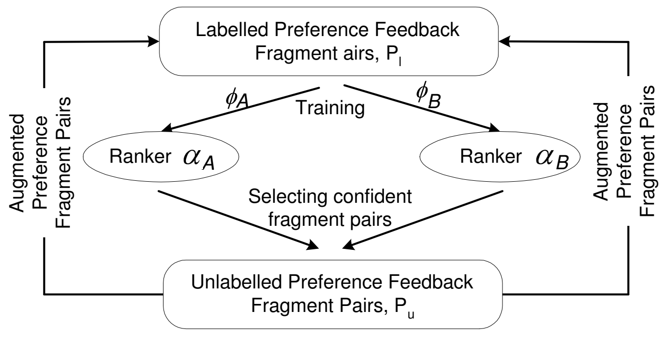

Labelled Preference Feedback Fragment airs, P_l
ϕ_A
Training
ϕ_B
Augmented Preference Fragment Pairs
Ranker α_A
Ranker α_B
Augmented Preference Fragment Pairs
Selecting confident fragment pairs
Unlabelled Preference Feedback Fragment Pairs, P_u

Figure 13. The underlying idea of the RSCF algorithm

The guideline for partitioning the feature vector, $\phi(Q, F)$, is that, after the partition, each subvector must be independent and sufficient for the later ranking. We now show our two subvectors in Figure 14

$$\phi_A = \{Rank(DAT,1),\ Rank(DAT,5),\ Rank(DOC,1),\ Rank(DOC,5),\ Rank(DFT,1),\ Rank(DFT,5),\ Rank(CUS,1),\ Rank(CUS,5),\ Sim_K,\ Sim_E,\ Grn_{Sib},\ Grn_{Tag},\ Grn_{Att}\}$$

$$\phi_B = \{Rank(DAT,3),\ Rank(DAT,10),\ Rank(DOC,3),\ Rank(DOC,10),\ Rank(DFT,3),\ Rank(DFT,10),\ Rank(CUS,3),\ Rank(CUS,10),\ Sim_A,\ Sim_P,\ Sim_O,\ Grn_{Chi},\ Grn_{Dis_+},\ Grn_{Dis_-}\}$$

Figure 14. The subvectors used in co-training

In general, the number of rankers used in the co-training framework can be more than two. However, it is difficult to identify enough features from the labelled fragments to generate more than two feature subvectors that are rich enough to train the corresponding rankers. Even if the feature vector is large in dimensions, we still should use a feature selection algorithm to eliminate some unimportant features and to reduce the dimensions of the feature vector so that the training process can be more efficient. In our study, we choose two rankers that adopt ϕ_A and ϕ_B for running the co-training algorithm.

We now introduce the parameter *prediction error*, which is a common criterion used to evaluate the performance of a classifier in *machine learning.* We use prediction error to evaluate the performance of the rankers in our experiments.

Definition 4.1 (Prediction Error) *Let us call a linked fragment pair, F_i and F_j, ($F_i \neq F_j$)* disconcordant *if the pair has different rankings in two given lists [28]. Given a ranker, α, trained with RSVM, the* prediction error *of α is defined as the percentage of the linked fragment pairs in the original test data set that are* disconcordant *according to α after an iteration.*

For example, if there are fifty fragment pairs in the original training data set and among them fifteen fragment pairs are *disconcordant* links according to α after some iterations, the prediction error of this iteration would be 0.3. We also need to define another new concept, *prediction difference* Δ, as the terminating criterion used in the RSCF algorithm. Δ is defined as the percentage of *disconcordant* links between α_A and α_B. For example, α_A and α_B select ten preference fragment pairs from the unlabelled preference fragment pairs and add them into the labelled data set. If five selections from α_A are ranked differently according to α_B, then Δ is equal to 0.5.

The RSCF algorithm is presented as Algorithm 3. At each iteration, the unlabelled preference fragment pairs that have the largest difference between their projection on $\overrightarrow{\omega}$ (i.e., the most confident ones) are selected and then added into the preference pair set. We finally obtain one enlarged set of labelled fragment pairs, P_l, and two rankers, α_A and α_B, as the output of the algorithm. We also set the threshold, τ, for Δ as an input to the algorithm and compute Δ at the end of each iteration. When Δ reaches τ, the whole process is terminated. Using the P_l output from Algorithm 3, we are able to obtain a final ranker, α_C, by the RSVM algorithm. We combine the trained rankers, α_A and α_B, into the ranker, α_C, which predicts better relevance judgments than the original (untrained) rankers. The complexity of RSCF algorithm depends on the two RSVMs running two subvectors as stated in Lines 2 and 3 in Algorithm 3. As RSVM based on approximation runs in polynomial time (recall the use of a non-negative slack variable in Algorithm 2), RSCF is also polynomial, since other steps are linear in complexity. A more detailed study and discussion of MXR overhead and learning costs is given in Section 5.

5 Experiments

In this section we report on experiments that study three aspects related to our RSCF-based MXR. First, we show that the training based on the RSCF technique is effective. Second, we compare the effectiveness of all the proposed rankers. Finally, we examine the overheads for loading our corpus and ranking retrieved fragments.

We present the results of experiments that we conducted over a collection of known XML datasets: DBLP, NASA, Shakespeare, Weblog, Treebank and Swissprot. Each dataset has its own features: DBLP is the popular bibliog-

Algorithm 3. RSCF Algorithm

```
Input:      $P_l$: An initial set of labelled fragment pairs;
            $P_u$: An initial set of unlabelled fragment pairs;
            $\tau$: The threshold for $\Delta$;
Procedure:
  1: while there exist preference fragment pairs without labels and $\Delta < \tau$ do
  2: Use RSVM to build $\alpha_A$ using features in $\phi_A$ of $P_l$;
  3: Use RSVM to build $\alpha_B$ using features in $\phi_B$ of $P_l$;
  4: Select the most confident unlabelled fragment pairs from $P_u$ according to $\alpha_A$
     and add them to $P_l$;
  5: Select the most confident unlabelled fragment pairs from $P_u$ according to $\alpha_B$
     and add them to $P_l$;
  6: Compute $\Delta$;
  7: end while
Output: $P_l$, $\alpha_A$ and $\alpha_B$.
```

raphy database and is relatively regular. NASA contains a mixture of data-centric and document-centric features. Shakespeare is a corpus of marked-up Shakespeare plays, containing much textual data. Weblog is the logging history of a web server, which is a regular data-centric document. Treebank is a large collection of parsed English sentences from the Wall Street Journal with deep recursive structures. SwissProt describes DNA sequences with a minimal level of redundancy.

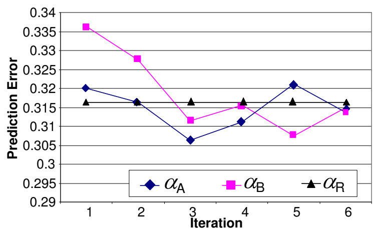
5.1 Effectiveness of the RSCF algorithm

We now show that the RSCF algorithm is effective in ranking XML fragments by measuring the prediction errors (recall Definition 4.1). We study the effectiveness in terms of the number of iterations and vary the number of queries in a training set and the number of XML labelled fragments as feedback.

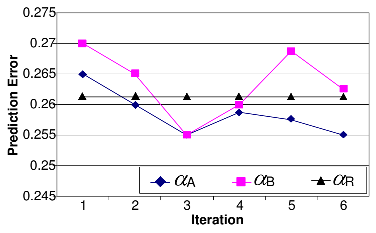
In the experiments, we compare the subrankers, α_A and α_B, which are implemented by the RSCF algorithm and trained on the two feature subvectors, ϕ_A and ϕ_B, as shown in Figure 14, with the ranker, α_R, which is implemented by the original RSVM algorithm and trained on the feature vector, ϕ. The user selects one to three preference pairs out of the top ten results at each iteration. We use the three training sets of ten, twenty and fifty queries and study the prediction errors of six iterations in RSCF.

Figures 15(a) to 15(c) show the prediction errors when the number of training queries are ten and the number of labelled fragments are one, two, and three, respectively. We can see that, prior to the process of learning, the RSVM ranker, α_R always outperforms the two subrankers, α_A and α_B. This is expected, since without labelled fragments the features of ϕ_A and ϕ_B are not as effective as ϕ. However, the effect of using the RSCF algorithm is shown after very few iterations, since the prediction errors of ϕ_A and ϕ_B decrease in the second and third iterations. After the fourth iteration, the prediction error becomes unstable or increases, since the remaining unlabelled fragments may have a higher probability of lower quality and, more likely, make the training collection worse. We also find that the average prediction error over the six iterations in Figure 15(a) is 0.32, while the arrange prediction error in Figure

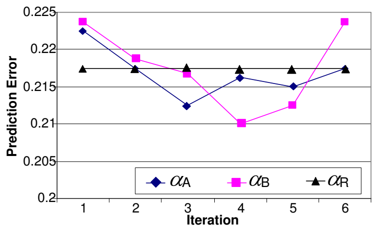
15(c) is 0.22. This also indicates the advantage of using the RSCF algorithm in this setting, since a few labelled XML fragments also help to improve the effectiveness. Roughly speaking, ϕ_A preforms slightly better than ϕ_B, since the former focuses more on the features of higher-ranked fragments than does the latter.



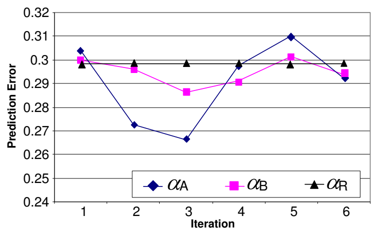
(a) 10 queries, 1 feedback sample



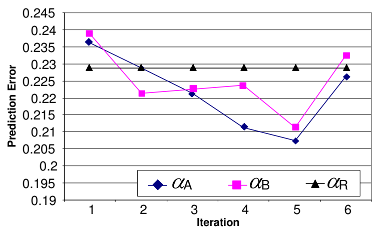
(b) 10 queries, 2 feedback samples



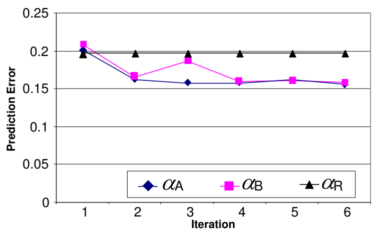
(c) 10 queries, 3 feedback samples



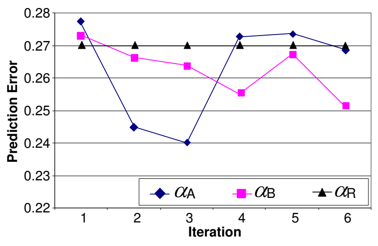
(d) 20 queries, 1 feedback sample



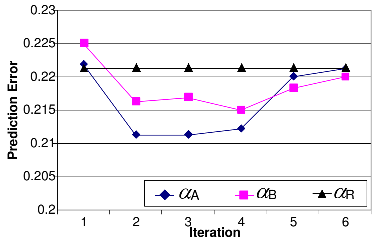
(e) 20 queries, 2 feedback samples



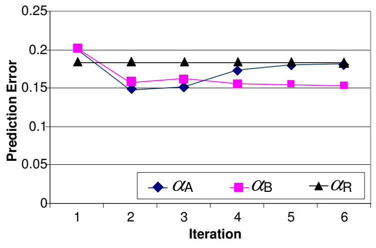
(f) 20 queries, 3 feedback samples



(g) 50 queries, 1 feedback sample



(h) 50 queries, 2 feedback samples



(i) 50 queries, 3 feedback samples

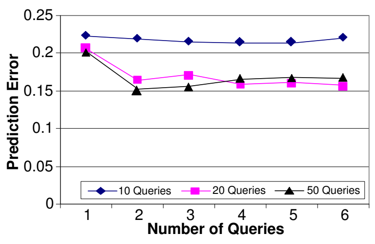
Figure 15. Prediction errors of various numbers of queries and feedback samples

We ran similar experiments with a larger set of queries. Figure 15(d) to 15(f) and Figure 15(g) to 15(i) show the results when the training queries are twenty and fifty. We find that the results are similar to the case of ten queries. The prediction error reaches the minimum around the third iteration and after that more feedback only increases the prediction errors. This further illustrates the strength of the RSCF; that is, only a small set of training queries is required to obtain a low prediction error.

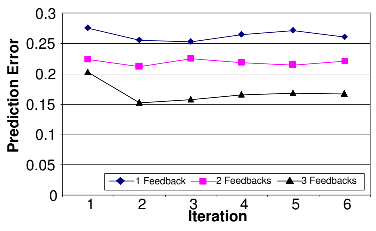
In further analyzing the above data, we compute the average prediction errors of α_A and α_B at different iterations and show three interesting results in Figures 16(a) to 16(c). First, Figure 16(a) shows that when increasing the number of training queries, a lower average prediction error can be obtained. However, the improvement is less when we compare the result of the cases of twenty and fifty queries with three labelled fragments. The average prediction errors are around 0.16 in both twenty- and fifty-query lines. A similar phenomenon can also be found when we make the same comparison with one and two labelled fragments. On the other hand, when we increase the number of labelled fragments for the case of fifty queries, we can see that the prediction error decreases in all three cases of one to three fragments, as shown in Figure 16(b). A similar phenomenon can also be found when we make the same comparison with the cases of ten and twenty queries. Finally, Figure 16(c) shows the overall improvement between the ranker, α_{C_i}, which has been trained by the RSCF algorithm and combines the rankers α_A and α_B against i labelled fragments ($1 \leq i \leq 3$), and α_R, which is the ranker obtained from standard RSVM. We can see that α_{C_i} outperforms α_R in terms of prediction errors.

5.2 Effectiveness of Rankers

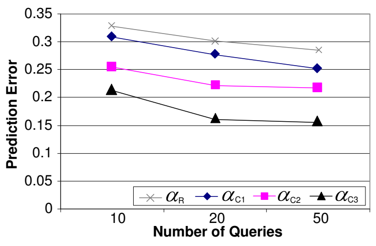
In this subsection, we first describe experiments that show the effectiveness of the four rankers that are developed according to the *DOC, DAT, DFT* and



(a) Average of α_A and α_B, 3 feedback samples



(b) Average of α_A and α_B, 50 queries



(c) Comparison of α_{C_i} and α_R, $(1 \leq i \leq 3)$

Figure 16. Further analyses of the prediction errors of the rankers

CUS ranking schemes presented in Section 3.4. Then, we demonstrate that, in general, the MXR has better search quality than the underlying search engines.

We performed extensive experimentation with the four XRs and the RSCF-based MXR. All the ranker, loader and searcher components are implemented by Java language, and their connection with Oracle is via JDBC. The loader is responsible for building the index and transfering the XML data into the underlying databases. The searcher consists of a retriever and a ranker. The retriever simply submits an SQL statement to the Oracle DBMS and obtains the fragments. Then, the ranker computes the rank for the fragments according to four different ranking schemes of *DOC*, *DAT*, *DFT* and *CUS*. Finally, the MXR assigns different weights to the rankers and generates a combined result in a round-robin manner (with duplicates removed).

The RSCF algorithm is implemented on the Ranking SVM, which is an open source code available in [34]. Besides this, the remaining work of the RSCF algorithm is mainly to deal with the input and output of the Ranking SVM algorithm, which is indeed straightforward to implement. The challenging part of the study is more the evaluation than the implementation. First, much effort is required to carry out relevance justification and compute the precision for all the rankers and the queries. Second, a lot of thought is needed to design another set of search queries that range from simple key-tags to complex key-tags.

The experiments are conducted on a Solaris 2.8 with CPU 1x300Mhz Ultra30 and 256MB memory. We then analyze the results obtained in these experiments based on the metric of *k-precision*, which is defined as

$$k - precision = \frac{\text{Number of top } k \text{ relevant results}}{k},$$

where k is the number of top results returned by the rankers. We can systematically judge which results are relevant according to a classification method to assign weights to the sample XML documents. If the retrieved fragment belongs to either of the sample XML documents, we assign it with the weighting specified in Figure 17. If there are n relevant results in the top k results, the precision is n/k. We first compare the four rankers regarding their particular strengths for handling different types of documents as follows. We test two different sets of queries, with each set consisting of 30 queries (see Appendix I). For example, when we submit a query that has many words, we expect the ranker to rank document-centric fragments higher than data-centric ones. For the CUS ranking, we assign the weight for each query using random numbers.

XML Documents	Data-centric Preference	Document-centric Preference
DBLP	1.0	0.0
Shakespeare	0.0	1.0
Weblog	1.0	0.0
Treebank	0.3	0.7
Swissprot	1.0	0.0
NASA	0.5	0.5

Figure 17. XML documents with data-centric and document-centric weightings.

The three values of worst precision, average precision and best precision are superimposed on Figures 18 and 19 for data-centric and document-centric queries. The relevance of the results is measured in an unbiased way by using precision. Precision and recall measures are common evaluation metrics used in information retrieval theory for searching textual documents [4]. However, we do not define the notion of the recall parameter in this context, since it is not practical to estimate the total number of relevant fragments on the XML datasets. Furthermore, the total number of fragments retrieved by the rankers is immaterial as the users are only concerned about a minority of fragments that is ranked at the top of a search result.

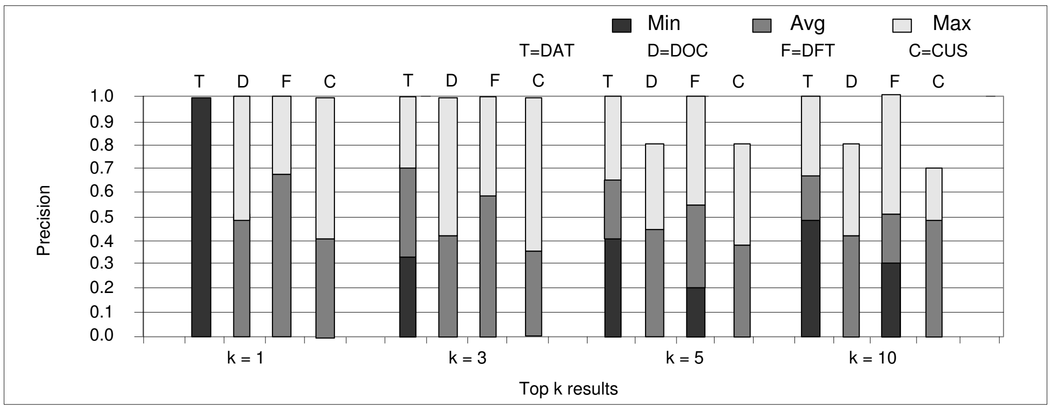


Figure 18. Comparison of the precision of the XRs when using the data-centric preference for judging relevance

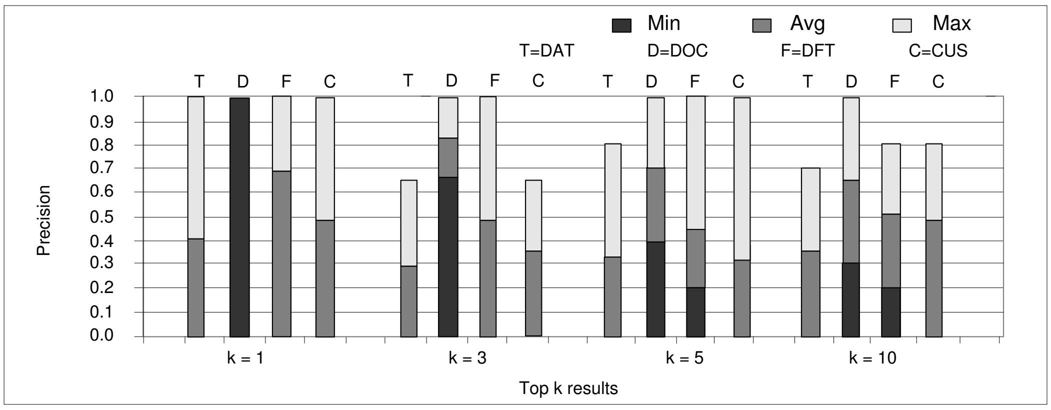


Figure 19. Comparison of the precision of the XRs when using the document-centric preference for judging relevance

We now first compare the four rankers in order to study their particular strengths in handling different types of documents. Both data-centric and document-centric rankers, DAT and DOC, outperform the other two XRs (DFT and CUS) when searching for their own types of queries (i.e., one of the preferences), which is consistent with our expectations. However, when the data-centric (document-centric) ranker is used to query document-centric (data-centric) documents, it may perform worse than the custom ranker, CUS, in which random numbers are assigned as weights. We also notice that the default ranker, DFT, performs in a stable manner in both types of queries. When we are querying data-centric documents, when k is within 3, we can see that all four XRs can return a maximum precision and the overall precision is better than querying document-centric documents. This is due to the fact that data-centric documents are regular in structure. Even a custom ranker with random assigned weights may still obtain perfect results. This means that even a naive user who has no idea of how to assign weights may still get satisfying results.

We then compare the precision of the RSCF-based MXR with the four underlying XRs in order to study the effectiveness of the RSCF technique. We test all the rankers with the thirty queries shown in Appendix I, which range from simple key-tags to complex key-tags with a wildcard, "$*$", in either the tag component or the word component of the key-tag. We evaluate the precision and analyze the results. In these experiments, we split the feature vector, ϕ into ϕ_A and ϕ_B for co-training purposes in the RSCF algorithm as already shown in Figure 14.

Figure 20 shows the comparison of the precision of MXR and the four XRs. We take the average precision obtained in the data-centric and document-centric preferences. It is clear that the RSCF-based MXR outperforms the four XRs. The reason for this is that the MXR is able to combine the strengths of various rankers. Note that the minimum and average precisions of the MXR are approximately 0.33 and 0.7 when k is within the top 10. This is obviously a great improvement over any individual ranker. An interesting finding is that when the set of queries consists of data-centric and document-centric types, the default ranker, DFT, performs better than the data-centric and document-centric rankers, DAT and DOC, and, in general, our system-defined ranker performs better than randomly assigned weights in the custom ranker.

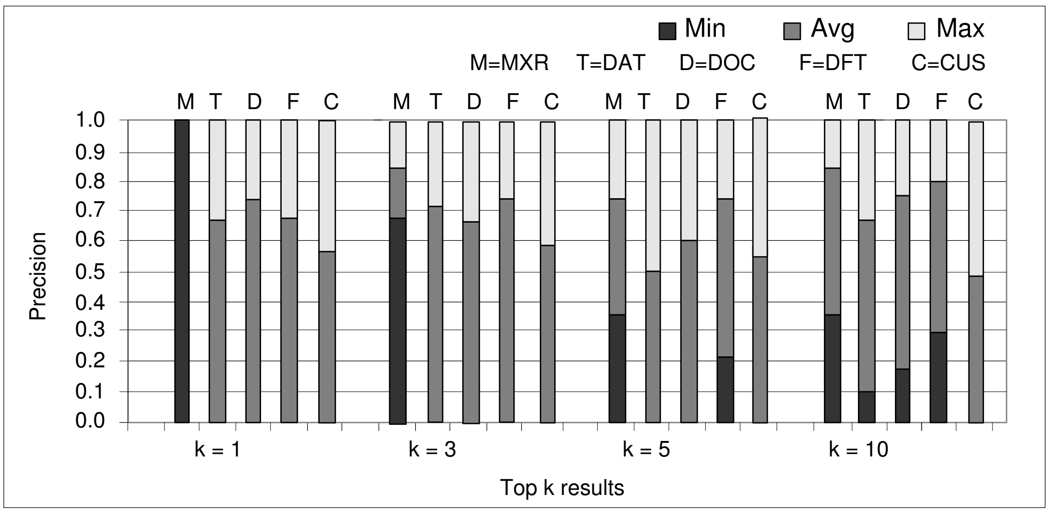


Figure 20. Comparison of the precision of the XRs and the RSCF-based MXR

5.3 Some Overhead Issues

In this subsection, we compare the running time of both ranked and unranked searching with RSCF-based MXR against the search queries derived from the benchmark XML documents in order to study the ranking overheads.

The loading time and the total size of the corpus that contains the six benchmark XML documents are 5611s and 482MB. Note that the preprocessing can be done off-line and once the documents are loaded, we can execute queries without loading the document for each query. We use the same set of search queries related to the benchmark data and simulate the text search by converting the key-tag pair into two input search keywords. For example, the key-tag "$\langle title\rangle XML\langle /title\rangle$" is converted into the keywords "title" and "XML" for carrying out a simple text search. We test the three kinds of searching with the ten queries from each XML dataset shown in the Appendix II, which also range from simple key-tags to complex key-tags. Figure 21 shows (1) the average running time of a simple text search, (2) the query time of our system before ranking, and (3) the running time of our MXR system (excluding the user feedback time) for the query in each workload.

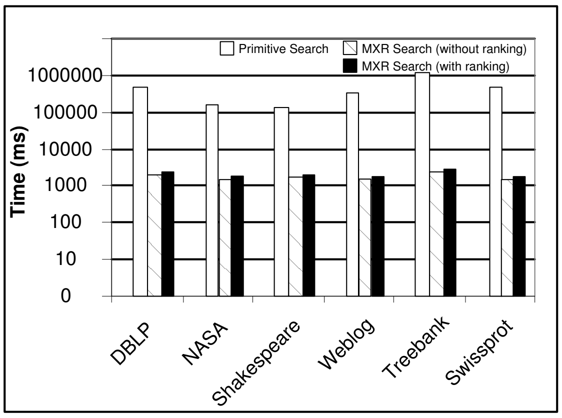


Figure 21. Comparison of processing time of simple text search, no-ranking search, and RSCF-based MXR search

As shown in Figure 21, the running time of the simple text search varies with different document sizes and the counterparts of our system remain roughly constant. The reason is that our RSCF-based MXR system requires a one-off preprocessing time for building the index and storing the XML data into the database. The total processing time in RSCF-based MXR depends on both the searching and ranking times. The computation of the ranking incurs some overhead ranging from 8% to 14% of the total processing time as shown in the last column in Figure 22. As we can see in the second and third columns, the searching time does not simply depend on the size of the document in the corpus. For example, Weblog requires much less searching time than does Treebank, although they are similar in size. This is due to the fact that Weblog is more regular in structure but Treebank has many more distinct elements and a deeper structure for indexing. It is also worth mentioning that the high ranking overhead for NASA is due to the fact that although its smaller size reduces the searching time, the returned fragments are large and thus more time is needed for ranking than in the cases of Weblog and SwissProt. Despite the ranking overhead, the searching time is, in general, still far better than simple text searching, as shown in the first bar of Figure 21.

XML Queries	Document Size (MB)	Searching time (ms)	Ranking time (ms)	Total time (ms)	MXR Ranking Overhead (%)
DBLP	134	3833	473	4306	10.98
NASA	25	2879	434	3313	13.10
Shakespeare	32	3608	317	3925	8.08
Weblog	89.8	3340	289	3629	7.96
Treebank	86	4275	459	4734	9.70
Swissport	114.8	3232	309	3541	8.73

Figure 22. Searching Time and Ranking Overhead

6 Concluding Remarks

We have presented a simple and effective approach to handling XML searching. Our proposed approach deals with the diversity of XML data in reality and the need for specifying target information in simple queries. We suggest that a search query can be expressed as a list of key-tags, which is a natural generalization of keywords in traditional searching. We have presented an extension of the vector space model that integrates various similarity measures between a search query and XML fragments.

As XML documents are diverse, we consider four ranking schemes based on different combinations of useful features and develop four XRs to compare their effectiveness. Our proposed XRs cater to different search needs. In order to adapt the XRs further, we develop an MXR and propose a training framework called RSCF, which is able to improve the retrieval quality via learning from the user's preference feedback in a progressive manner. Based on a co-training framework, our RSCF algorithm requires only a small set of labelled data for the training and does not intervene in the searching process. We demonstrate by an extensive set of experiments that the RSCF-based MXR indeed improves the retrieval quality when compared to the individual XRs.

In our prototype, we adopt a minimal indexing scheme to support the searching. First, each tag in the XML document is assigned with a numerical "tagID". Indexes are built as trees using this tagID. When the system processes a path (in XPath format), the path is translated into its corresponding numerical form and then we perform a search using the tree structure. We may consider this tree structure as the numerical representation of the XML scheme and it maintains the hierarchical structure of the XML document. Using the numerical representation should not make it difficult to apply existing advanced indexing techniques such as Compact Tree [40] and XSeq [32], to improve the performance of the searching.

There are still several issues that deserve further study in order to improve the performance of the MXR in searching XML data. In particular, an interesting direction is to apply our technique to cater to individuals' needs, in addition to adapting the search engine to a community of users. We believe that RSCF can be directly used to personalize search engines if the personal clickthrough data [28,35] can be specifically recorded. In this study, the search engines are studied as "standalone systems". From the point of view of applicability, we need to deploy the system in the context of modern Web applications. It is useful to study the issues concerning how to adopt the search engines in different services and how the engines are interoperable in a Web architecture, which may support mixed key-tag query and native XML query processing, such as in the recent work reported in [22]. Finally, it is worth carrying out a user study to check if the simple text search via key-tag queries is adequate for XML searching in practice. For example, we can provide subjects with a list of search tasks against a given XML corpus. This study would help us understand real user queries if the users are allowed to use only key-tags for searching.

Acknowledgement: We would like to express our sincere thanks to the editor and the reviewers, who gave very insightful and encouraging comments. This work is supported in part by grants from the Research Grants Council of Hong Kong, Grant Nos. HKUST6185/02E, HKSUT6165/03E and DAG04/05.EG10.

References

[1] S. Amer-Yahia, L. Lakshmanan, P. Shashank. *FleXPath: Flexible Structure and Full-Text Querying for XML.* In: Proc. of SIGMOD, 2004.

[2] S. Amer-Yahia, C. Botev and J. Shanmugasundaram. *TeXQuery: A FullText Search Extension to XQuery.* In Proc. of WWW, 2004.

[3] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava and D. Toman. *Structure and Content Scoring for XML.* In: Proc. of VLDB, 2005.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-wesley-Longman, 1999.

[5] B. Bartell, G.Cottrell, and R. Belew. *Automatic combination of multiple ranked retrieval systemss.* In Proc. of the 17th ACM SIGIR, p. 173–181, 1994.

[6] D. Beeferman and A. Berger. *Agglomerative clustering of a search engine query log.* In Proc. of the 6th ACM SIGKDD, p. 407–416, 2000.

[7] K. Bennet and A. Demiriz. *Semi-supervised support vector machines.* Advances in Neural Information Processing Systems, 11:368–374, 1998.

[8] A. Blum and T. Mitchell. *Combining labeled and unlabeled data with co-training.* In Proc. of the 11th annual conference on Computational learning theory, pp. 92–100, 1998.

[9] J. Bosak. *Shakespeare in XML.*
In: http://www.ibiblio.org/xml/examples/shakespeare/, 2004.

[10] J. Boyan, D. Freitag, and T. Joachims. *A machine learning architecture for optimizing web search engines.* In Proc. of the AAAI workshop on Internet-Based Info. Sys., 1996.

[11] D. Carmel, Y. S. Marrek, M. Mandelbrodand, Y. Mass, A. Soffer. *Searching XML documents via XML fragments.* In: Proc. of the 26th ACM SIGIR, 2003.

[12] J. Clark and S/ DeRose. *XML path language (XPath) version 1.0. W3C Recommendation.* http://www.w3.org/TR/xpath, 1999.

[13] S. Cohen, J. Mamou, Y. Kanza, Y. Sagiv. *XSEarch: A Semantic Search Engine for XML.* In: Proc. of the VLDB, 2003.

[14] S. Cohen, R. Shapire, and Y. Singer. *Learning to order things.* Journal of Artificial Intelligence Research, Vol. 10, p. 243–270, 1999.

[15] E. Curtmola, S. Amer-Yahia, P. Brown and M. Fernandez. *GalaTex: A Conformant Implementation of the XQuery FullText Language.* In Proc. of XIME-P Workshop, 2005.

[16] CWI Database Groups. *MonetDB/XQuery.* http://monetdb.cwi.nl/XQuery/.

[17] European Bioinformatics Institute. *The UniProt/Swiss-Prot Protein Knowledgebase.* http://www.ebi.ac.uk/swissprot/index.html, 2004.

[18] N. Fuhr. *Optimum polynomial retrieval functions based on the probability ranking principle.* ACM Trans. on Info. Sys., Vol. 7, Issue 3, p. 183–204, 1989.

[19] N. Fuhr. *Probabilistic Models in Information Retrieval.* The Computer Journal, Vol. 35, No. 3, p. 243–255, 1992.

[20] N. Fuhr and K. Grossjohann. *XIRQL: An Extension of XQL for Information Retrieval.* ACM SIGIR 2000 Workshop on XML and Information Retrieval, pp. 172–180, 2000.

[21] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. *XRANK: Ranked keyword search over XML documents.* In: Proc. of the ACM SIGMOD, 2003.

[22] A. Halverson, et al. *Mixed Mode XML Query Processing.* In: Proc. of VLDB, 2003.

[23] B.O. Huberman. *The laws of the Web.* Cambridge, MA: MIT Press, 2003.

[24] *Initiative for the Evaluation of XML Retrieval..* http://inex.is.informatik.uni-duisburg.de:2004.

[25] U. Ilhan. *Application of K-NN and FPTC based text categorization algorithms to turkish news reports.* Bilkent University, Dept. of Comp. Eng., Technical Reports, 2001.

[26] H.V. Jagadish, et al. *TIMBER: A Native XML Database* VLDB Journal 11(4), 2002.

[27] T. Joachims. *Evaluating retrieval performance using clickthrough data.* In Proc. of the ACM SIGIR Workshop on Mathematical/Formal Methods in IR, 2002.

[28] T. Joachims. *Optimizing search engines using clickthrough data.* In Proc. of the 8th ACM SIGKDD, p. 133–142, 2002.

[29] Jon M. Kleinberg. *Authoritative sources in a hyperlinked environment.* Journal of the ACM, Vol. 46, Issue 5, p. 604–632, 1999.

[30] M. Ley. *Digital Bibliography & Library Project.* In: http://dblp.uni-trier.de/, 2004.

[31] Y. Li, C. Yu and H.V. Jagadish. *Schema-Free XQuery.* In: Proc. of VLDB, 2004.

[32] X. Meng, Y. Jiang, Y. Chen and H. Wang. *XSeq: An Indexing Infrastructure for Tree Pattern Queries.* In: Proc. of SIGMOD, 2004.

[33] L. Page, S. Brin, R. Motwani and T. Winograd. *The pagerank citation ranking: Bringing order to the web.* Technical report, Stanford Digital Library Technologies Project, 1998.

[34] SVM^{light}. *Support Vector Machine.* http://svmlight.joachims.org/, Ver 6.01 2004.

[35] Q. Tan, X. Chai, W. Ng, and D. L. Lee. *Applying Co-training to Clickthrough Data for Search Engine Adaptation.* In Proc. of the 9th DASFAA, LNCS Vol. 2973, page 519-532, 2004.

[36] A. Theobald and G. Weikum. *The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking.* In Proc. of EDBT, pp. 477–495, 2002

[37] *XML SQL Utility in Oracle.* http://www.oracle.com/index.html, 2004.

[38] World Wide Web Consortium. *XQuery 1.0: An XML Query Language.* http://www.w3.org/TR/xquery/, W3C Working Draft 22 August 2003.

[39] World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Full-Text.* http://www.w3.org/TR/2005/WD-xquery-full-text-20050404/, W3C Working Draft 4 April 2005.

[40] Q. Zou, S. Liu and W.W. Chu. *Ctree: A Compact Tree for Indexing XML Data.* In: Proc. of WIDM, 2004.

Appendix I: The following are the set of queries that we used in the experiments in Section 5.2. They can be classified into five groups: Q1-Q5 are simple queries (one key-tag) with complete key-tags; Q6-Q10 are simple queries having wildcards in some tag components; Q11-Q15 are simple queries having wildcards in some keyword components; Q16-Q20 are complex queries (more than one key-tag) with complete key-tags; Q21-Q22 are complex queries having wildcards in some tag components; Q23-Q25 are complex queries having wildcards in some keyword components; Q26-Q30 are complex queries having wildcards in both tag and keyword components.

The query set used for querying data-centric documents for the experiment shown in Figure 18:

(1) <year>2001</year>

(2) <author>Lam</author>

(3) <X>20</X>

(4) <title>data</title>

(5) <title>XML</title>

(6) <∗>XML</∗>

(7) <∗>1998</∗>

(8) <∗>Lam</∗>

(9) <∗>Antony</∗>

(10) <∗>get</∗>

(11) <PP>∗</PP>

(12) <weblog>∗</weblog>

(13) <date>∗</date>

(14) <author>∗</author>

(15) <act>∗</act>

(16) <year>2001</year>, <title>XML</title>

(17) <play>mark</play>, <act>mark</act>

(18) <weblog>get</weblog>, <date>.1999</date>

(19) <PP>X</PP>, <title>X</title>, <act>X</act>

(20) <date>2001</date>, <url>.edu</url>, <author>Al</author>

(21) <∗>200</∗>, <∗>199</∗>

(22) <∗>Al</∗>, <∗>La</∗>

(23) <play>∗</play>, <act>∗</act>

(24) <author>∗</author>, <speech>∗</speech>

(25) <datasets>∗</datasets>, <history>∗</history>

(26) <play>philo</play>, <∗>mark</∗>, <speech>∗</speech>

(27) <author>Al</author>, <year>∗</year>, <∗>X</∗>

(28) <weblog>∗</weblog>, <∗>get</∗>, <S>200</S>

(29) <year>19</year>, <date>∗</date>, <∗>XML</∗>, <∗>database</∗>, <∗>query</∗>

(30) <act>∗</act>, <speech>Dear</speech>, <speaker>mark</speaker>, <speaker>philo</speaker>, <∗>hi</∗>

The query set used for querying document-centric documents for the experiment shown in Figure 19:

(1) <line>as I told you</line>

(2) <play>His name is Licio</play>

(3) <persona>Lord</persona>

(4) <weblog>.html</weblog>

(5) <title>XML Query</title>

(6) <∗>management system</∗>

(7) <∗>let me see</∗>

(8) <∗>.</∗>

(9) <∗>Licio</∗>

(10) <∗>get</∗>

(11) <description>∗</description>

(12) <history>∗</history>

(13) <date>∗</date>

(14) <author>∗</author>

(15) <speaker>∗</speaker>

(16) <line>as I told you</line>, <speaker>Lucentio</speaker>

(17) <speech>my Lord</speech>, <speaker>mark</speaker>

(18) <description>proper motion</description>,
<tablehead>Declination</tablehead>

(19) <field>difference between</field>, <field>Number of accepted observations
</field>, <definition>Proper motion</definition>

(20) <_BACKQUOTES_>456</_BACKQUOTES_>,
<_COMMA_>++</_COMMA_>, <_PERIOD_>==</_PERIOD_>

(21) <∗>Hi</∗>, <∗>0</∗>

(22) <∗>between</∗>, <∗>result</∗>

(23) <play>∗</play>, <act>∗</act>

(24) <author>∗</author>, <speech>∗</speech>

(25) <datasets>∗</datasets>, <history>∗</history>

(26) <speaker>mark</speaker>, <∗>my lord</∗>, <line>∗</line>

(27) <play>∗</play>, <title>∗</title>, <∗>king</∗>

(28) <initial>D</initial>, <para>get</para>, <footnote>Cape</footnote>

(29) <description>∗</description>, <definition>∗</definition>,
<reference>∗</∗>, <title>Standard Stars</title>

(30) <act>∗</act>, < *speech* >Dear</*speech* >, <speaker>mark</speaker>,
<speaker>philo</speaker>, <∗>hi</∗>

The query set with a mix of data-centric and document-centric documents for the experiment shown in Figure 20:

(1) <year>2001</year>

(2) <author>Lam</author>

(3) <speaker>Mark Antony</speaker>

(4) <play>Mark Antony</play>

(5) <title>XML</title>

(6) <∗>XML</∗>

(7) <∗>1998</∗>

(8) <∗>Lam</∗>

(9) <∗>Mark Antony</∗>

(10) <∗>get</∗>

(11) <PP>∗</PP>

(12) <weblog>∗</weblog>

(13) <date>∗</date>

(14) <author>∗</author>

(15) <act>∗</act>

(16) <year>2001</year>, <title>XML</title>

(17) <PP>X</PP>, <title>X</title>, <act>X</act>

(18) <date>2001</date>, <url>.edu</url>, <author>Al</author>

(19) <field>difference between</field>, <field>Number of accepted observations
</field>, <definition>Proper motion</definition>

(20) <_BACKQUOTES_>456</_BACKQUOTES_>,
<_COMMA_>++</_COMMA_>, <_PERIOD_>==</_PERIOD_>

(21) <∗>200</∗>, <∗>199</∗>

(22) <∗>between</∗>, <∗>result</∗>

(23) <play>∗</play>, <act>∗</act>

(24) <author>∗</author>, <speech>∗</speech>

(25) <datasets>∗</datasets>, <history>∗</history>

(26) <play>philo</play>, <∗>mark</∗>, <speech>∗</speech>

(27) <author>Al</author>, <year>∗</year>, <∗>X</∗>

(28) <weblog>∗</weblog>, <∗>get</∗>, <S>200</S>

(29) <description>∗</description>, <definition>∗</definition>,
<reference>∗</∗>, <title>Standard Stars</title>

(30) <act>∗</act>, <speech>Dear</speech>, <speaker>mark</speaker>,
<speaker>philo</speaker>,<∗>hi</∗>

Appendix II: The following are the set of queries that we used in the experiments in Section 5.3.

The query set used for querying DBLP for the experiment shown in Figure 21:

(1) <year>2001</year>

(2) <author>Lam</author>

(3) <title>XML</title>

(4) <∗>XML</∗>

(5) <∗>1998</∗>

(6) <author>∗</author>

(7) <date>∗</date>, <∗>1998</∗>

(8) <date>2001</date>, <url>.edu</url>, <author>Al</author>

(9) <year>2001</year>, <title>XML</title>

(10) <year>19</year>, <date>∗</date>, <∗>XML</∗>, <∗>database</∗>, <∗>query</∗>

The query set used for querying NASA for the experiment shown in Figure 21:

(1) <year>19</year>

(2) <identifier>xml</identifier>

(3) <field>D</field>

(4) <∗>xml</∗>

(5) <∗>1998</∗>

(6) <identifier>∗</identifier>

(7) <date>∗</date>, <∗>1998</∗>

(8) <author>*</author>, <*>XML</*>

(9) <date>2001</date>, <field>remark</field>, <altname>I</altname>

(10) <date>19</date>, <field>∗</field>, <∗>XML</∗>, <∗>remark</∗>

The query set used for querying Shakespeare for the experiment shown in Figure 21:

(1) <line>as I told you</line>

(2) <play>His name is Licio</play>

(3) <persona>Lord</persona>

(4) <∗>let me see</∗>

(5) <∗>Licio</∗>

(6) <author>∗</author>

(7) <speaker>mark</speaker>, <∗>my lord</∗>, <line>∗</line>

(8) <line>as I told you</line>, <speaker>Lucentio</speaker>

(9) <speech>my Lord</speech>, <speaker>mark</speaker>

(10) <act>∗</act>, < *speech* >Dear</*speech* >, <speaker>mark</speaker>, <speaker>philo</speaker>, <∗>hi</∗>