Configurable Indexing and Ranking for XML Information Retrieval

Shaorong Liu, Qinghua Zou and Wesley W. Chu
UCLA Computer Science Department, Los Angeles, CA, USA 90095
{sliu, zou, wwc}@cs.ucla.edu

ABSTRACT
Indexing and ranking are two key factors for efficient and effective XML information retrieval. Inappropriate indexing may result in false negatives and false positives, and improper ranking may lead to low precisions. In this paper, we propose a configurable XML information retrieval system, in which users can configure appropriate index types for XML tags and text contents. Based on users' index configurations, the system transforms XML structures into a compact tree representation, Ctree, and indexes XML text contents. To support XML ranking, we propose the concepts of "weighted term frequency" and "inverted element frequency," where the weight of a term depends on its frequency and location within an XML element as well as its popularity among similar elements in an XML dataset. We evaluate the effectiveness of our system through extensive experiments on the INEX 03 dataset and 30 content and structure (CAS) topics. The experimental results reveal that our system has significantly high precision at low recall regions and achieves the highest average precision (0.3309) as compared with 38 official INEX 03 submissions using the strict evaluation metric.

Categories and Subject Descriptors
H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval models – *retrieval models*.

General Terms
Algorithms, Experimentation, Ranking

Keywords
XML Information Retrieval, XML Indexing, XML Ranking

1. INTRODUCTION
As the World Wide Web (WWW) is becoming a major means of disseminating and sharing information, there has been an exponential increase in the amount of data in web-compliant formats such as HyperText Markup Language (HTML) and Extensible Markup Language (XML). XML is essentially a textual representation of the hierarchical (tree-like) data where a meaningful piece of data is bounded by matching starting and ending tags, such as <name> and </name>. Due to its simplicity and expressiveness, XML has become the most popular format for information representation and data exchange on the web.

To cope with the tree-like structures in the XML data model, a great deal of research has been conducted to provide flexible and effective retrieval methods in the Information Retrieval (IR) community [2-5, 10, 11]. The INitative for the Evaluation of XML Retrieval (INEX) [13], for example, was established in April, 2002 and has prompted XML researchers worldwide to promote the evaluation of effective XML retrieval.

Compared with traditional IR, XML information retrieval has introduced many new challenges. For example, traditional IR only focuses on content only (CO) queries, while XML information retrieval supports both CO queries and content and structure (CAS) queries. CAS queries enable users to specify queries more precisely than traditional CO queries, but introduce new challenges of indexing XML structures for efficient retrieval. In addition, traditional IR has only one data type, i.e., plain text, while XML may contain data of various types, such as plain text, numbers, date and time. Thus we need multiple content processing methods and indexing types for the heterogeneous contents in XML documents to support various search predicates. Further, not all tags in an XML document are semantically meaningful [1]. Improper indexing of non-semantic tags can result in false negatives. Thus, we need a user-configurable framework to differentiate semantic tags from non-semantic tags, such as tags used for presentation purposes only. Finally, an XML query result may not always be an entire document. It can be any deeply nested XML element, i.e., dynamic document [10]. Therefore, the traditional static document ranking method is no longer sufficient for ranking XML query results. As a result, we need a new ranking method.

In this paper, we address the above challenges as follows:

First, to support efficient processing of CAS queries, we transform XML document trees into a compact indexing tree, Ctree. Ctree provides both path summaries and detailed element-to-element relationships in the XML document trees. Thus it can answer most structured queries very efficiently without accessing the original XML documents.

Second, to support various search predicates over XML documents with heterogeneous tags and contents, we propose a configurable XML information retrieval system. In this system, XML documents are first "scanned" to collect structure and content statistics for each group of similar elements. These statistics are then stored in spreadsheets and presented to users. Users can then select tag index types, content processing operations and index types for each group of similar elements based on the statistics.

SIGIR'04, July 25–29, 2004, Sheffield, South Yorkshire, UK.

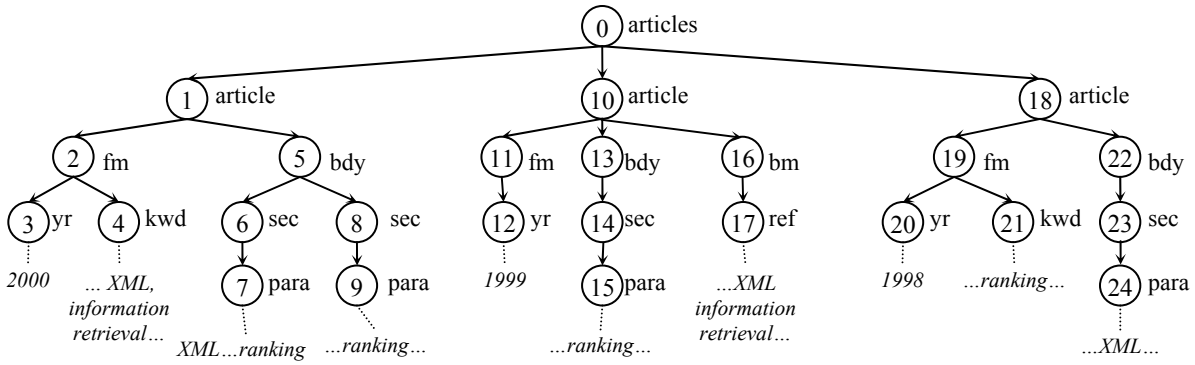


Figure 1: An example of an XML document tree.

Third, to support the dynamic document concept in XML, we extend the classic Vector Space Model (VSM) [9] in traditional IR to the XML model and propose the concepts of "weighted term frequency" and "inverted element frequency," where the weight of a term depends on its frequency, popularity and its location in an XML document.

To empirically evaluate the effectiveness of our system, we have conducted experiments on the INEX 03 dataset with all the 30 CAS topics. Our experimental results reveal that our system has significantly high precision at low recall regions and has the highest average precision (0.3309) compared with all the 38 official INEX submissions using the strict evaluation metric.

The rest of the paper is organized as follows. In section 2 we introduce the XML data model and Ctree. Section 3 presents the configurable indexing framework as well as the five value index types. In sections 4 and 5, we describe our query processing and ranking methods. Section 6 contains the experimental studies that evaluate the effectiveness of our system. Section 7 reviews related work. We conclude our work in section 8.

2. BACKGROUND

2.1 XML data model

We model an XML document as an ordered, labeled tree where each element (attribute) is represented as a node and each element-to-subelement (or element-to-attribute) relationship is represented as an edge between the corresponding nodes. We assume that each node is a triple (*id*, *label*, <*value*>), where *id* uniquely identifies the node, *label* is the name of the corresponding element or attribute, and *value* is the corresponding element's text content or attribute's value. *Value* is optional because not every element has a text content. We consider an attribute as a sub-element of an element and a reference IDREF as a special type of value.

For example, Figure 1 shows a sample XML document tree with 25 nodes numbered from 0 to 24. Each circle represents a node with the node *id* inside the circle and *label* beside the circle. To distinguish text contents from element (attribute) nodes, the value of a node is linked to the node by a dotted line.

We now introduce the definitions for *label path* and *equivalent nodes* which are useful for describing Ctree in Section 2.2.

Definition 1 (**Label Path**) A label path for a node *v* in an XML document tree is a list of dot-separated labels of the nodes on the path from the root node to *v*.

For example, node 12 in Figure 1 can be reached from the root node through the path: node 0→10→11→12. Thus the label path for node 12 is *articles.article.fm.yr*.

Definition 2 (**Equivalent Nodes**) Two nodes in an XML document tree are equivalent if their label paths are the same.

For example, nodes 3 and 12 in Figure 1 are equivalent because their label paths are both *articles.article.fm.yr*.

2.2 Ctree

Indexing the structures of XML documents is very important for efficient processing of structured XML queries. Many current indexing methods create indices only on the predefined nodes (e.g., [4]), such as leaf level nodes. Such approaches are simple and efficient, but sometimes may not be flexible enough to support queries with any structure constraint and to retrieve nodes at any level.

To overcome this problem, we transform an XML document tree *D* into a compact indexing tree, Ctree [12], which is a two-level bidirectional tree: group level and element level. The group level provides path summaries for *D* and contains edges from parent groups to their child groups. The element level provides detailed element-to-element relationships and has links pointing from child elements to their corresponding parent elements.

Similar to most path index approaches (e.g., DataGuide [6]), the first step in Ctree construction is to cluster equivalent nodes in *D* into groups. There is an edge linking from group A to group B if the label path of group A is the longest prefix of that of group B. For example, the path summary for the XML document tree (Figure 1) is shown in Figure 2a. Each group is represented as a dotted box with its label above the box. The numbers inside each dotted box are the node identifiers from Figure 1. For instance, nodes 3, 12, and 20 in Figure 1 share the same label path and thus they are in the same group *yr* in Figure 2a.

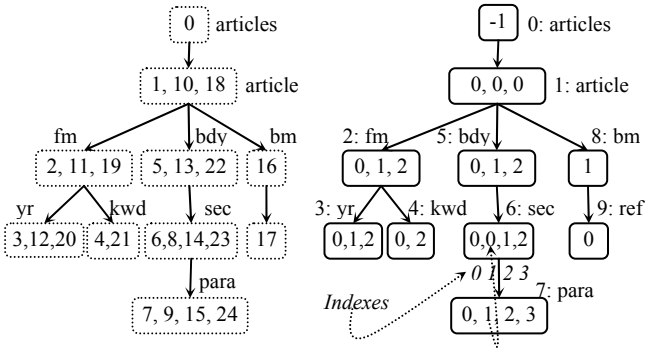
As shown in past research, such path summaries greatly facilitate the evaluation of simple path expressions (i.e., path expressions with a single branch and without filters) by searching only relevant parts of the tree. For example, for a query (Q_1) */articles/article/bdy/sec*, the path summary in Figure 2a implies that all the nodes in group *sec* are the answers since their label paths match Q_1. Such path summaries, however, are insufficient for answering non-simple path expressions due to their incompleteness. They do not preserve the hierarchical relationships among individual nodes in an XML document tree. For example, with the path summary in Figure 2a, we cannot determine the hierarchical relationships

between node 19 in group *fm* and node 21 in group *kwd*. Such relationships, however, are important in answering non-simple path expressions. For example, for a query (Q_2): */articles/article /fm[kwd]*, the path summaries in Figure 2a indicate that nodes in group *fm* are candidate answers. We cannot, however, determine which node in group *fm* can answer Q_2 unless the hierarchical relationships between the individual nodes in group *fm* and those in group *kwd* are provided.

Therefore, the second step in Ctree construction is to order the nodes in a group into a list by their corresponding preorders in *D*. We shall call the nodes in a Ctree group as elements for differentiating them from the nodes on an XML document tree. The elements in a group list are accessible by their corresponding indexes. The index for an element *e* in a group *g* is its relative order in *g*. Instead of storing node identifiers in each group, Ctree stores the indexes of these elements' corresponding parent elements in the parent group of *g*. Since the root element in the root group has no parent, we set the value in the root group to -1.

Figure 2b shows the corresponding Ctree for the XML document tree in Figure 1, where each box represents a group with its *id* and *label* above the box. The numbers inside each box are the values in the group list. For simplicity, we use the notation *g:e* to represent an element in group *g* with index *e*. For example, nodes 9 and 8 in the path summaries (Figure 2a) correspond to the elements *g:e* = 7:1 and *g:e* = 6:1 in the Ctree (Figure 2b). The Ctree in Figure 2b implies that 6:1 is the parent of 7:1 since the value of 7:1 is 1 and the parent group of group 7 is group 6.

With the Ctree in Figure 2b, we can answer not only simple but also non-simple path expressions very efficiently without accessing the original XML document tree. For example, the values in group *kwd* imply that elements *g:e* = 2:0 and *g:e* = 2:2 are the answers for Q_2.



(a) Path summary (b) Ctree

Figure 2: The path summary and the Ctree for the XML document tree in Figure 1.

3. INDEX CONFIGURATION

3.1 Motivation

To illustrate the importance of index configurations in supporting various search predicates over XML documents, we present a fragment of a sample XML article in Figure 3. The example is simple as compared with most XML scientific articles, but has many characteristics of XML in document-processing applications, including semantic (e.g., <article>) and non-semantic tags (such as presentation-purpose tags, e.g., <*scp*>), and annotations (tags and their embedding text) about notations, corrections or clarifications (e.g., <*note*> ...</*note*>).

```
01 <article>
02 <author> Webb </author>
03 <title> A K<scp>NOWLEDGE</scp> Based Web Data
04.      Integration and Exchange </title>
05.   <year> 2003 </year>
06.   <pages> 180 -200 </pages>
07.   <body>
08.   <sec>The problem of information integration
09.      <note>see  reference 2</note> and exchange …
10.   </sec>
11.   </body> </article>
```

Figure 3: An example XML document.

To illustrate how matching keyword and phrases interacts with XML tags and annotations, we consider the following two examples. Suppose a user is interested in articles about "knowledge," the article in Figure 3 is relevant since its title contains that word. However, if we do not ignore the tags <*scp*> and </*scp*>, the article in Figure 3 will not be returned because <*scp*> separates "K" from "NOWLEDGE". For another example, suppose that a user wants to find articles with a section containing the phrase "information integration and exchange," the article will be judged as irrelevant if we do not ignore the annotation element *note*. Therefore, to support keyword and phrase searching in XML documents, we should allow users to ignore non-semantic tags and annotations during the indexing.

Besides tag heterogeneity, XML document contents also contain data of various types including plain texts, numbers, dates and times. The heterogeneous contents in an XML document require diverse value processing operations before indexing and multiple value index types. Inappropriate processing operations can lead to undesirable results. For example, removing stop words and stemming non-stop words are applicable to the text contents of an article's title, body or section, but not to the text content of an article's author. For example, if we stem the text between <author> and </author> in Line 2, the element *author* will be returned as relevant to a CO query "web, internet" since the stem of "webb," which is text content of element *author*, is "web." To avoid such undesirable results, we shall allow users to configure proper content processing operations before indexing.

Therefore, we propose a configurable index framework that allows a user to specify tag index types, content processing operations and content index types.

3.2 Index configuration framework

3.2.1 System architecture

Figure 4 shows the architecture for the configurable XML information retrieval system, which performs two types of functions: document indexing and query evaluation.

Indexes for a collection of XML documents can be built in the following three steps. First, XML documents are sent to a *Scan* module to collect statistics about the structure and content characteristics of the XML documents. These statistics are then stored in an Excel spreadsheet and presented to a user. Second, based on the statistics, a user configures index options for each group of equivalent nodes. Finally, based on the index configurations, the *Index Builder* correspondingly constructs a

Ctree and builds value indexes. For complex datasets, such as the INEX dataset, a user may not be familiar with the dataset characteristics even with the collected statistics. In this case, a user can either use the default index options or leverage on the index configurations provided by a domain expert.

The *Query Evaluation* evaluates the incoming query based on the indexes, and then ranks the results based on users' ranking configurations to obtain a list of ranked results.

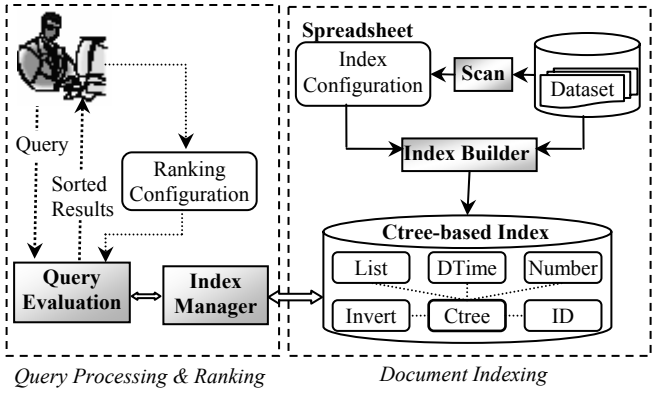


Figure 4: The XML information retrieval system architecture.

3.2.2 Index configuration

Many XML datasets, such as the INEX dataset, do not have schema with specific data types for each element and attribute. To facilitate index configurations for such datasets, we use a *Scan* module to collect statistics of a dataset while parsing it. Since equivalent nodes in XML document trees share similar characteristics, the *Scan* module collects the structure and content statistics for each group of equivalent nodes.

With the collected statistics, a user can specify a tag index type, a set of value processing operations and a value index type for each group of equivalent nodes, as described in the following:

- Tag Index Type: *Index* or *No Index*. The tag index type allows users to indicate whether to keep or ignore the tag during indexing.
- Value Processing Operations: 1)*TokenType*: whether to select digit, word, mixed or all tokens in the value for indexing; 2) *IsStopping*: whether to remove stop words; 3) *IsStemming*: whether to use stemming functions; and 4) *IsToLower*: whether to transform a text to its lower cases.
- Value Index Type: 1) *No Index*; 2) *Invert*; 3) *Number*; 4) *DTime*; 5) *List* or 6) *ID*. *No Index* means that the values for the nodes in this group will be ignored during indexing. The latter five value index types will be explained in Section 3.3.

If an XML dataset contains too many groups of equivalent nodes, such as the INEX dataset which contains 13262 distinct groups, then its configuration can be based on each group of nodes with the same label. For instance, there are only 204 distinct labels in the INEX dataset.

3.3 Value index types

The heterogeneous contents in XML documents require multiple value index types. Thus we propose five value index types: *Invert*, *List*, *Number*, *DTime* and *ID*, defined as follows, to support values of common XML data types, such as *xs:string* and *xs:decimal*, as defined in the XML schema and some special data values such as values for IDREF attributes.

- *Invert Type*: treats a value as a bag of tokens and maps a token to a list of elements.
- *List Type:* treats a value as a whole without further breaking it into tokens and maps a value to a list of elements.
- *Number Type:* maps a numeric value to a list of elements. Furthermore, a B+-index is created on top of (number, element) pairs to support numerical range searches.
- *DTime Type:* maps a value of date or time type to a list of elements. Similarly, we create a B+-index on (time, element) pairs to support range searches.
- *ID Type:* indexes IDREF attribute values and maps a referring element to a referred element.

Each value index type supports a common search function:

List search (value, gid?)

That is, given a value predicate and a group identifier, the *search* function returns a list of elements satisfying the value predicate in the group. If the group identifier is not specified, the *search* function returns a list of elements in any group that satisfy the search predicate.

4. QUERY EVALUATION

4.1 Query format & model

We use the INEX 03 query format [7], which is based on a subset of XPath path expressions [14] with an addition of an *about* predicate. A path expression contains a sequence of nodes connected with axes and some nodes may have value predicates, i.e., filters. The last node in a path expression is called a target node and its matches are retuned as query results. The syntax of an *about* predicate is *about(path, string)*, which specifies certain contexts (i.e., *path*) to be about a specific content (i.e., *string*). The *string* parameter may contain a set of terms separated by spaces, where a term is either a single word or a phrase in double quotes. Furthermore, query term modifiers, such as "+" and "-," are introduced to facilitate users to specify preferences and rejections over certain terms. For example, suppose that a user is interested in articles about XML and information retrieval, published between 1999 and 2000, and with sections preferred to be about ranking, the query can be formulated as Q_3: *//article[(./fm//yr = '2000' OR ./fm//yr = '1999') AND about(., 'XML "Information Retrieval"')]//sec[about(., '+ranking')]*.

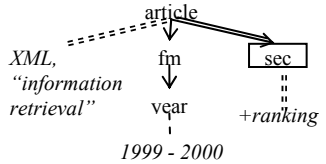


Figure 5: The tree representation of Q_3.

Similar to the tree representation of XML documents, we also represent queries as trees: nodes in path expressions become the nodes in trees. Axes are represented as edges between the corresponding nodes

with a single arrow for a "/" axis and a double arrow for a "//" axis. Filters are represented as value predicates on the corresponding nodes. To distinguish *about* predicates from other normal XPath value predicates, *about* predicates are linked to their corresponding nodes with double dotted lines, while normal value predicates are linked to their corresponding nodes with single dotted lines. Finally, a target node is emphasized with a box. For example, Figure 5 illustrates the tree representation of Q_3.

4.2 Query processing

After transforming an XML query into a tree representation, we can evaluate the query based on Ctree structure and value indexes in the following three steps as shown in Figure 6.

Input: T, a Ctree with value index
T_Q, a query tree
Output: R, a ranked list of elements
1 An empty list →R
2 Locate frames: Ctree groups→query nodes
3 For each frame F
4 ⋰ Evaluate value predicates;
5 ⋮ Combine results for each query node to a ranked list, R';
6 ⋱ Merge R' to R;
7 Return R

Figure 6: The Ctree-based query processing algorithm.

First, the algorithm locates a set of frames. A frame F is a set of Ctree groups such that each group in F matches a node in the query tree T_Q and that these groups as a whole match the query's tree structure (Line 2). For example, there is one frame consisted of groups (1, 2, 3, 6) in the Ctree (Figure 2b) for Q_3, where (1, 2, 3, 6) matches the query nodes (*article*, *fm*, *year*, *sec*) respectively.

Second, for each frame F, the algorithm evaluates each value predicate on the elements in a Ctree group that is assigned to a query node with the corresponding value predicate (Line 4). Normal value predicates are processed in a Boolean way. For example, for the predicate $1999 \le$ *//article/fm/year* ≤ 2000 in Q_3, the elements in group 3 are either relevant or non-relevant. Irrelevant elements are pruned directly. An a*bout* predicate, however, is processed in a non-Boolean way and we will discuss its evaluation in detail in Section 5.

Third, for each frame F, the algorithm combines the evaluation results for each query node in the query based on the element-to-element relationships in a Ctree. The results from F are sorted into a ranked list, denoted as R' (Line 5).

Finally, the results from each frame are merged and ranked based on their retrieval status value (RSV) (Line 6). A RSV indicates a result's relevancy to the query. Line 7 outputs a ranked result list.

5. RESULT RANKING

In this section, we present how to calculate the RSVs for query results from a given frame. We first present how to calculate the RSVs for elements from a single *about* predicate and then discuss how to combine the RSVs from multiple *about* predicates.

Before presenting our ranking scheme, we first define two user configurable parameters as follows:

$\sigma(x)$: weight for a label x. $\forall x, \sigma(x) \ge 0$ and the default value is 1.

$\theta(m)$: weight for a query term modifier m ("", "+", or "-"). $\theta(\text{"+"}) \ge \theta(\text{""}) > 0$ and $\theta(\text{""}) > \theta(\text{"-"})$.

5.1 Weighted term frequency

Due to the hierarchical nature of XML, the content of an element e is also considered part of the content of any e's ancestor element. This introduces the challenge of how to calculate the relevancy of a given term t within a certain element e, where t could appear in any element nested within e. For example, for the *about(article, 'XML "information retrieval"')* predicate in Q_3, the term '*XML*' can be in a keyword, paragraph or reference part of an article. The occurrences of a term t in different sub-elements of e have different importance. For example, 'XML' in a keyword part of an article is more important than an "XML" in a paragraph, which in turn is more important than an "XML" in a reference part. As a result, it may be inaccurate if we simply count the frequency of a term t within element e without considering the locations of term t in element e.

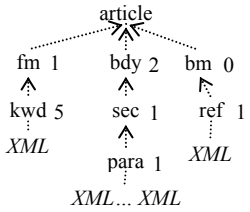


Figure 7: A weighted term frequency example.

Therefore we introduce the concept of "weighted term frequency," which assigns high weights to terms in important locations and low weights to terms in unimportant locations. For a given element e, we can identify the location of a specific sub-element e' containing a term t by the "relative" label path from e to e'. For example, in Figure 7, under the element *article*, the term 'XML' appears once, twice and once in the relative paths *fm.kwd*, *bdy.sec.para* and *bm.ref* respectively.

Since the number of distinct locations in an XML dataset can be very large, it is laborious to assign weights for all possible locations. The number of distinct labels, however, is usually small. Therefore we can estimate the weight (or importance) of a location (relative label path) $l = x_1 \ldots x_s$ by a function of $\sigma(x_i)$ $(1 \le i \le s)$, $f(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_s))$, with the following properties:

1) $f(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_s))$ is a monotone increasing function with regard to any $\sigma(x_i)$ $(1 \le i \le n)$.
2) $f(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_s)) = 0$ if any $\sigma(x_i) = 0$ $(1 \le i \le n)$.

The first property is straightforward and the second property is due to the semantics of $\sigma(x_i) = 0$, which implies that a user is not interested in terms occurring under element x_i. Therefore any term in the text content of either element x_i or element x_j $(i \le j \le n)$ is not relevant.

One straightforward implementation of $f(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_s))$ is to estimate the weight of a relative label path l as the product of the weights of the labels in l:

$$w(l) = \prod_{i=1}^{s} \sigma(x_i), \text{ where } x_i \text{ is a label of } l$$

For an element e in a given Ctree group g, where g is assigned to a query node associated with an *about* predicate with a term t, let $tf(g, e, l_i, t)$ be the term frequency of t in a location l_i under element e in group g. We can estimate the weighted term frequency of term t within element e in a group g, denoted as $tf_w(g, e, t)$, as follows:

$$tf_w(g,e,t) = \sum_{j=1}^{r} w(l_j) \cdot tf(g,e,l_j,t)$$

where r is the number of locations under element e in group g containing term t.

For example, in Figure 7, the weights for each label are shown beside the label. For instance, σ(kwd) =5. The weights for the relative label paths *fm.kwd*, *bdy.sec.para*, and *bm.ref* are 5*1=5, 2*1*1=2 and 0*1=0 respectively; and the frequencies of term "XML" in *fm.kwd*, *bdy.sec.para*, and *bm.ref* are 1, 2 and 1 respectively. Therefore, the weighted term frequency of 'XML' in element *article* is 5*1+2*2+0*1=9.

5.2 Inverted element frequency

Terms with different popularities in XML elements have varying degrees of discriminative power. It is well established in IR that a term frequency (*tf*) needs to be adjusted by a factor of inverse document frequency (*idf*). A very popular term with a small *idf* has less discriminative power than a rare tem with a large *idf*. Similarly, we propose the concept of "inverted element frequency," i.e., *ief*, to distinguish query terms in an *about* predicate with different discriminative powers. The inverted element frequency of term t within a specific group g can be measured as follows:

$$ief(g,t) = \log(\frac{|E_g|}{|E_g(t)|})$$

where $|E_g|$ is the total number of elements in group g assigned to a query node with an *about* predicate containing term t. $|E_g(t)|$ is the number of elements in group g containing t.

5.3 RSV from a single *about* predicate

Query terms with different modifiers in an *about* predicate are of different importance from a user's point of view. Thus, we introduce a user-configurable ranking parameter θ(m) that defines the weight for a query term modifier m. In general, the relationships among θ("+"), θ("") and θ("-") shall satisfy: 1) θ("+") ≥ θ("") > 0; and 2) θ("") > θ("-"). That is, a term prefixed with "+" is more important than, or at least as important as, a term without any prefixing modifier, which in turn is more important than a term with a "-." For example, θ("") = 1, θ("+") = 2, and θ("-") = 0.

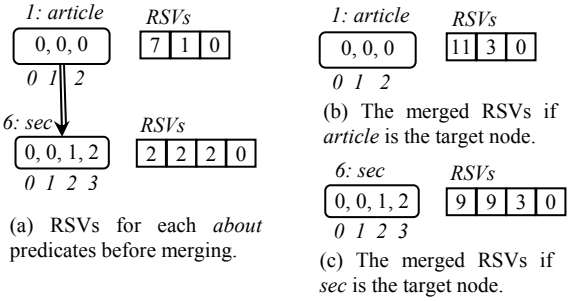
With the introduction of weighted term frequency, inverted element frequency and term modifier weight, we are now ready to define the RSV of an element e in a group g for an *about* predicate α as follows:

$$RSV(g,e,\alpha) = Max(\sum_{t_k \in \alpha} \theta(m(t_k)) \cdot tf_w(g,e,t_k) \cdot ief(g,t_k), 0)$$

where t_k is a term in α and θ(m(t_k)) is the weight for the query term modifier associated with term t_k. Since users might assign a negative value for θ("-"), which may lead to negative RSVs, we compare the weighted sum of tf_w*ief with 0 and select the max of the two to ensure that *RSV(g, e, α)* is always non-negative.

5.4 RSVs from multiple *about* predicates

Since a query Q may contain multiple *about* predicates, we need to merge the RSVs from each *about* predicate into the elements in the Ctree group assigned to the target node in Q. For example, Q_3 in Figure 5 has two *about* predicates: α_1 (*about(article, 'XML "information retrieval"')*) and α_2 (*about(sec, +'ranking')*). After processing α_1 and α_2, the elements in group *article* and group *sec* are associated with a list RSVs as shown in Figure 8a. For example, the RSV for element 1:0 to α_1 is 7, i.e., *RSV(1, 0, α_1)* =7.



(a) RSVs for each *about* predicates before merging.

(b) The merged RSVs if *article* is the target node.

(c) The merged RSVs if *sec* is the target node.

Figure 8: An example of merging multiple RSVs to the elements in the group assigned to the target query node.

Depending on the location of the target node in the query tree, RSVs are either merged from a descent group to an ancestor group or from an ancestor group to a descendant group. Thus, there are two cases to consider:

1) Merging results from a descendant group to an ancestor group

If the results are merged from a descendant group g_D containing elements relevant to an *about* predicate α_i to an ancestor group g_A containing elements relevant to another *about* predicate α_j, the RSVs of elements in group g_A are updated according to:

$$RSV(g_A, e_A, \alpha_i \,\&\, \alpha_j) = RSV(g_A, e_A, \alpha_j) + \sum_{l=1}^{n} RSV(g_D, e_{D_l}, \alpha_i)$$

where e_A is an element in g_A, e_{D_l} is an element in group g_D and also a descendant of e_A, and n is the number of elements in group g_D that are descendant of e_A.

For example, suppose that *article* is the target node in Q_3 instead, according to the formula above, the updated RSVs for the elements in group 1 are shown in Figure 8b. For instance, RSV(1, 0, α_1&α_2) = RSV(1, 0, α_2) + RSV(6, 0, α_1) + RSV(6, 1, α_1) = 7+2+2 = 11 because elements 6:0 and 6:1 are descendants of element 1:0.

2) Merging results from an ancestor group to a descendant group

If the results are merged from an ancestor group g_A containing elements relevant to an about predicate α_j to a descendant group g_D containing elements relevant to another about predicate α_i, the RSVs of elements in group g_D are updated according to:

$$RSV(g_D, e_D, \alpha_i \,\&\, \alpha_j) = RSV(g_D, e_D, \alpha_i) + RSV(g_A, e_A, \alpha_j)$$

where element e_D in group g_D is a descendant of element e_A in group g_A.

For example, since *sec* is the target node in Q_3, according to the formula above, the updated RSVs for the elements in group 6 are shown in Figure 8c. For instance, RSV(6, 0, α_1&α_2)= RSV(6, 0, α_1) + RSV(1, 0, α_2) = 2+7 = 9 because element 1:0 is an ancestor of element 6:0.

6. EXPERIMENTAL STUDIES

We used the INEX 03 dataset and CAS topics to evaluate the effectiveness of our configurable XML information retrieval system. For the INEX 03 CAS topics, there are two tasks: strict CAS (SCAS) and vague CAS (VCAS). A query's structure must be strictly matched in SCAS, while it can be vaguely matched in VCAS. In this paper, we only focus on SCAS task. We implemented the configurable XML information retrieval system in C# and ran the experiments on a 2.8GHz PC with 1G of RAM running Windows XP.

6.1 Runs

We have conducted a set of experiments with the same set of index configurations but different weight configurations. Due to space limit, the indexing configurations used for the experiments can be downloaded from our website [15]. Also we only list two weight configurations A and B in Table 1. Tags indexed but not listed below are assigned with the default weight 1. The main difference between configuration A and B is that the weight for the tag *bm* is 0 in B, which prunes articles with only references or appendixes about topic of interest.

Table 1 Tag weight configurations A and B.

	bm	Fm	bdy	atl	abs	kwd	st
A	1	3	1	3	1	2	3
B	0	5	1	5	1	3	5

We tested three runs with the above two tag weight configurations and the same set of query term modifier weights: θ("+")=1.8 and θ("")=1. Document components containing query terms prefixed with "-" in a query are judged as irrelevant to the query. All the precision/recall curves are plotted with the INEX on-line tool.

1. **SCAS-65-A**: This run evaluated CAS topic 65 with configuration A and the results are shown in Figure 9a.
2. **SCAS-65-B**: This run evaluated CAS topic 65 with configuration B and the results are shown in Figure 9b.
3. **SCAS-ALL**: This run evaluated all the 30 CAS topics with configuration B and the results are shown in Figure 10.

6.2 Results evaluation and analysis

To evaluate the relevancy of an XML document component to a topic, the INEX 03 working group proposed a two-dimension relevancy metric (exhaustiveness, specificity). Two methods are proposed to quantize the metric into a single relevancy value between 0 and 1: strict and generalized quantization [8]. In a strict quantization, the relevancy metric is quantized to be either 1 or 0. In a generalized quantization, the relevancy metric is quantized to be 0, 0.25, 0.5, 0.75 or 1.

The average precision/recall curves for CAS topic 65 with the tag weight configurations A and B using the strict quantization are illustrated in Figure 9a and 9b respectively. We notice that the average precision for CAS topic 65 is quite high either with configuration A or B. Furthermore, our system has a very high precision at low recall regions. For example, with configuration A, the precision is 1 when the recall is less than 0.318; and with configuration B, the precision stays at 1 until the recall exceeds 0.518. From Figure 9, we can see that properly adjusting ranking configurations can significantly improve the precision/recall.

Figure 10 illustrates the average precision/recall curves for all the 30 CAS topics. The average precision for all 30 the CAS topics, 0.3309, is notably high and is about 4% improvement over the top 1 (0.3812) of all the 38 official INEX 03 submissions. More importantly, from Figure 10c, we can see that using the strict quantization, our system has the highest precision when recall is less than 0.5 as compared with all the 38 official INEX 03 submissions, which implies that our system is able to return the most relevant answers sooner than other systems. This is a very important feature since most users typically have only enough patience to browse the top few returned answers.

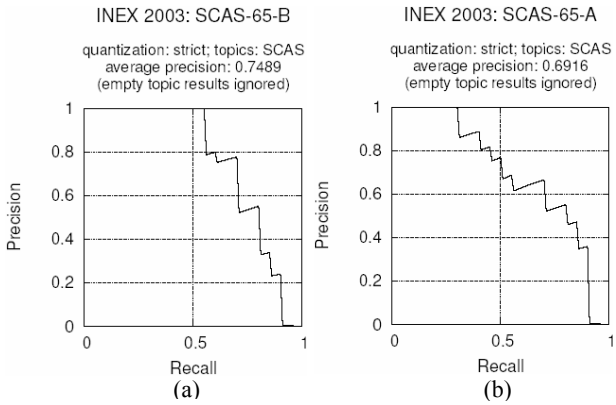


Figure 9: The precision/recall curves of SCAS-65-A and SCAS-65-B using strict quantization.

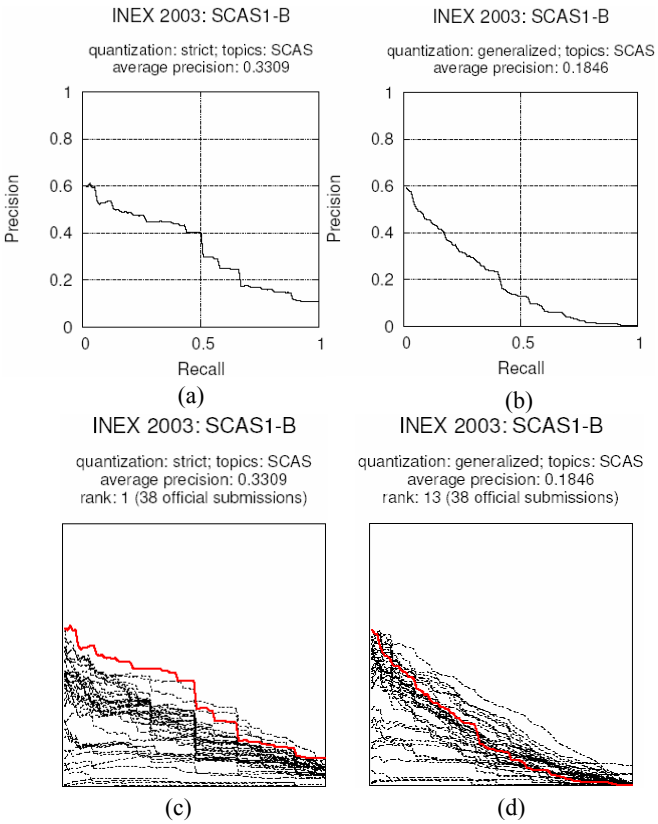


Figure 10: The average precision/recall curves of SCAS-ALL using both strict and generalized quantization.

By comparing Figure 10a and 10b, we note that our system performs better with the strict evaluation metric than with the generalized evaluation metric. This is because our system returns only a small number of results for each topic due to the strict implementation of an "AND" operator. For example, our system will not judge any article containing only 'XML' or only "information retrieval" as relevant to Q_3. Therefore, the number of our results for most CAS topics is less than 200 and many of them are even less than 50, while most other systems return about 1500 results for each CAS topic. One way to improve the precision with the generalized evaluation metric is to increase the result size by relaxing the strict interpretation of an "AND" operator.

7. RELATED WORK

There have been a number of studies investigating XML information retrieval [2-5, 10, 11]. Many of these studies can be classified into the following three categories: XML retrieval language, indexing, and ranking. We focus our review of related work on these three areas.

XML Retrieval Language

XIRQL [4] is the first full-functional XML information retrieval language proposed. It supports weighting and ranking, relevance-oriented searches, data types with vague predicates and semantic relativism. XIRQL is powerful but not simple enough for new users. To overcome this problem, [3] proposed the concept of XML fragments as queries. It is much simpler and allows users to specify term preferences and rejections.

XML Indexing

Most work on XML indexing extend the idea of "inverted index" for content indexing in traditional IR to support both content and structure indexing in the XML model. For example, in [4], nodes of predefined categories are indexed and associated with term statistics. Users are allowed to search at the level of indexing nodes or nodes that are of hierarchical combination of indexing nodes. [5] generalized [4] to support the retrieval of nodes at any granularity level by combining the statistics of those predefined nodes at query time. In [3] XML document collections are split into small "documents" based on the predefined elements. A vector of (term, context) pairs is extracted from each document. Indexes are created on these (term, context) pairs associated with statistics for ranking calculation.

XML Ranking

Many ranking approaches proposed so far leverage on the mature ranking models developed in traditional IR and extend them to the XML model. For example, in [4] a novel "augmentation" technique is proposed to retrieve nodes that are hierarchical combinations of indexing nodes. During the "augmentation," the statistics and inverted lists of indexing nodes are propagated upward in the document tree with lower term weights. [3] extends the traditional VSM where each unit is a single term to the XML model with each pair (term, context) as a basic unit.

8. CONCLUSION AND OUTLOOK

In this paper, we proposed a configurable XML information retrieval system, which allows users to configure proper index types for tags in an XML document to avoid false negatives. It also enables users to select appropriate content processing operations and index types for the heterogeneous XML text contents to avoid false positives. Further, we proposed a new XML ranking methodology based on the concepts of "weighted term frequency" and "inverted element frequency," where the weight of a term depends on its frequency and location in an XML element as well as its popularity among similar elements. We evaluated the effectiveness of our system through extensive experiments on the INEX 03 dataset and all the 30 CAS topics. Our experimental results reveal that 1) properly setting tag weights can significantly improve the precision; and 2) our approach has significantly high precision at low recall regions and achieves the highest average precision (0.3309) as compared with all the 38 INEX 03 official submissions using strict evaluation metric. Our future work includes extending current work to support CO and VCAS tasks.

ACKNOWLEDGEMENT

This work is supported by NSF Award ITR# 0219442.

9. REFERENCES

[1] S. Amer-Yahia, M. Fernandez, D. Srivastava and Y. Xu. Phrase Matching in XML. In *VLDB 2003*, pp. 177-188, 2003.

[2] R. Baeza-Yates, N. Fuhr and Y. Maarek. Second Edition of the XML and IR Workshop. In *SIGIR Forum*, Volume 36 Number 2, Fall 2002.

[3] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass and A. Soffer. Searching XML Documents via XML Fragments. In *Proceedings of SIGIR'03*, Toronto, Canada, 2003.

[4] N. Fuhr and K. GrossJohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In *Proceedings of SIGIR'2001*, New Orleans, LA, 2001.

[5] T. Grabs and H. J. Schek. Generating Vector Spaces On-the-fly for Flexible XML Retrieval. In [1].

[6] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *the Proceedings of VLD*B 1997, 1997.

[7] G. Kazai, M. Lalmas and S. Malik. INEX'03 Guidelines for Topic Development.

[8] G. Kazai, M. Lalmas and B. Piwowarski. INEX'03 Relevance Assessment Guide.

[9] G. Salton and M.J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, New York, 1983.

[10] T. Schlieder and H. Meuss H. Querying and Ranking XML Documents. In *Journal of American Society for Information Science and Technology*, Volume 53, Issue 6, pp. 489-503, 2002.

[11] A. Theobald and G. Weikum. Adding relevance to XML. In *the Proceedings of WebDB* 2000.

[12] Q. Zou, S. Liu and W. Chu. Ctree: A Compact Two-level Bidirectional Tree for Indexing XML Data. *UCLA-CS Technical Report* #TR040010, 2004.

[13] INitiative for the evaluation of XML Retrieval. http://qmir.dcs.qmul.ac.uk/INEX

[14] XPATH. http://www.w3.org/TR/xpath.

[15] http://fargo.cs.ucla.edu/inexdemo/inexsearch.aspx.