Efficient, Effective and Flexible XML Retrieval Using Summaries

M. S. Ali, Mariano Consens, Xin Gu, Yaron Kanza, Flavio Rizzolo, and Raquel Stasiu

University of Toronto
{sali, consens, xgu, yaron, flavio, raquel}@cs.toronto.edu

Abstract. Retrieval queries that combine structural constraints with keyword search are placing new challenges on retrieval systems. This paper presents *TReX*—a new retrieval system for XML. *TReX* can efficiently return either all the answers to a given query or only the top-k answers. In this paper, we discuss our participation in the annual Initiative for the Evaluation of XML Retrieval (INEX) workshop in the ad-hoc track. Our main contribution is to investigate the use of summaries and the flexibility they provide when dealing with structural constraints. We describe algorithms for retrieval using summaries. Experimental results are presented showing that *TReX* answers queries efficiently and effectively.

1 Introduction

Recent research efforts have combined the structured data management capabilities of databases with the powerful keyword search capabilities of information retrieval (IR) systems. One of the best known of these research efforts is the INEX [1] initiative. INEX is a forum dedicated to research in information retrieval from collections of XML documents. In XML retrieval, queries are combinations of keywords (content queries), structural hints (vague queries) and structural constraints (strict queries). Query responses are composed of XML document fragments (*i.e.,* specific elements) that satisfy the structural conditions and are returned ranked according to relevance criteria based on the content and structural components of the query.

To assess the *effectiveness* of the ranked answers returned by XML retrieval systems, human judgments are collected for the answers to standard queries, which are called topics, on XML collections. The collections are shared among all of the INEX participants. Based on the collections, INEX participants propose and agree on the topics for the human judges. System implementors develop their ranking criteria and assess the quality of the answers from their systems against the human judgments. Participants' ranking criteria generally use well-established IR techniques for content scoring that have been extended to incorporate the structural conditions specified in the topic. We refer to this extension as *structural scoring*. The XML retrieval community is just starting to develop an

understanding of structural scoring. We expect that in the coming years a wide range of different techniques will be proposed and assessed. To this effect, our efforts have concentrated on developing an XML retrieval system that supports *flexible* structural scoring. We believe that this will foster more experimentation and will help move forward the state-of-the-art over the long term as we begin to understand the different ways that structure is used in XML retrieval. Our contention is that XML retrieval systems must be capable of *efficiently* combining IR evaluation techniques with new structural ranking capabilities. There are still a wide spectrum of challenges to overcome. As an example, this is illustrated in the strict interpretation of structural constraints because these constraints have the same efficiency demands on the system as those placed on a structured XML query engine (*i.e.,* those posed on an XPath or XQuery capable processor). *TReX* is a step toward overcoming these challenges.

In this paper we describe the techniques used by the *TReX* system to support efficient, effective and flexible XML retrieval. *TReX* retrieves relevant XML fragments by simultaneously using indexes on paths in the XML (*summaries*) and indexes on keywords (*inverted lists*). Previous work has established the advantages of using summaries for structured XML queries [6]. This paper applies summaries to content and vague structure retrieval queries. Two methods for computing queries are considered. In the *exhaustive* method, queries are computed directly from the indexes. Our second method is meant for quickly computing the top-k answers to a query. It relies on the exhaustive method to first precompute and store lists of ranked elements for each query keyword and path expression. Then, the system employs the *threshold algorithm* (TA) for efficiently combining the ranks according to the keywords in the query. We provide experimental results showing the efficiency and the effectiveness of *TReX*'s use of summaries in support of flexible structural scoring in XML retrieval.

Several proposals in the literature extend the traditional keyword-style retrieval to the XML model [8, 13, 14]. Vague structural conditions were introduced in [23] and complemented with full-text conditions in [3, 4]. A query algebra for IR style processing of XML data was introduced in [5]. Although only for keyword queries, XRANK [13] is the only system that provides efficient support for finding the top-k results. Other recent proposals for XML ranked retrieval include [17] and [20]. The former uses dataguides and TA-style top-k algorithms [11], but differs from our work in that their experiments are limited to DB-like queries rather than XML retrieval queries. In contrast, [20] focuses on efficient evaluation of approximate structural matches without considering keyword search. The closest work to ours is TopX [24]. We follow the baseline top-k algorithm described in that work, but we do not use their probability predictor function nor invoke costly random access to resolve structural constraints. Our scoring model is similar to existing scoring models such as TopX [24] and BM25E [18]. The main difference from TopX and BM25E is that tags (element names) are the only structural constraints influencing the score whereas, in *TReX*, the scoring function uses more flexible summary-based constraints.

The structure of the paper is as follows. Section 2 introduces the retrieval queries supported by the *TReX* system. Section 3 introduces summaries. Section 4 describes the evaluation mechanisms used by *TReX*. Finally, Section 5 presents experimental evidence of the effectiveness and efficiency of *TReX*.

2 Retrieval Queries

TReX is designed for evaluating *NEXI* queries [25] over a given XML corpus. *NEXI* (*Narrowed Extended XPath I*) is a query language for specifying retrieval queries. It was devised and has been used in the context of the *Initiative for the Evaluation of XML Retrieval* (INEX)[19]. *NEXI* is built upon XPath [7]. On the one hand, it narrows XPath by excluding function symbols and some axes. On the other hand, it extends XPath with the function `about()`, which denotes a vague interpretation of its input. A *NEXI* query is composed of two types of constraints, structural and textual. The `about()` function can be applied to both. The structural constraints are expressed in XPath-like syntax and the textual constraints are keywords.

Example 1. Consider the following *NEXI* query
`//article[about(., XML retrieval)]//sec[about(., inverted list)]`.
This query specifies a search for sections that are relevant to the keywords "inverted list" that appear in articles that are relevant to "XML retrieval".

The answer to a query consists of elements that satisfy the structural and textual constraints. The elements, in an answer, are ranked according to their relevance to the search. In general, elements that contain the specified search terms should be ranked higher than elements that do not. For instance, the answer to the query in Example 1 are `sec` elements that are descendants of `article` elements, *i.e.,* elements that are in the answer to the XPath expression `//article//sec`. All `sec` elements in the answer should be ranked according to their relevance to the keywords "inverted" and "list", and the relevance of their ancestor `article` elements to the keywords "XML" and "retrieval".

The scoring function *TReX* uses is a version of the Okapi BM25 formula [10] modified for XML. The *TReX* function is a generalization of the scoring function employed in the TopX query engine [24]. Its novelty is that the score of an element is given with respect to a set S of elements specified by the structural constraints of the query. Before presenting the formula, we provide some necessary notation. We denote by $tf(t, e)$ the *term frequency* of the term t in the element e. This function returns the number of occurrences of t in the textual content of e, where the textual content is considered a bag of terms. We denote by $ef_S(t)$ the element frequency of a search term t, with respect to a set S of elements. This function returns the number of elements that contain t, among the elements in S. The *length* of an element e, denoted $length(e)$, is the number of words in the textual content of e. That is, $length(e) = \sum_{\{t|t \text{ is a term in } e\}} tf(t, e)$. Finally, we denote the size of a set S by $|S|$.

Given a list $t_1, \ldots, t_m$ of terms, an element set S and an element e in S, the BM25 score of e is given by the following formula.

$$score_s(e \mid t_1, \ldots, t_m) = \sum_{i=1}^{m} \frac{(k_1 + 1) \cdot tf(t_i, e)}{K + tf(t_i, e)} \cdot \log \left(\frac{|S| - ef_S(t_i) + 0.5}{ef_S(t_i) + 0.5} \right)$$

where

$$K = k_1 \left((1 - b) + b \cdot \frac{length(e)}{\text{avg}\{length(e') \mid e' \in S\}} \right)$$

Okapi BM25 was originally developed using statistics of all documents in the corpus. In the context of XML, BM25 has been modified to use statistics at the granularity of elements. One may note that any scoring function based on term-frequency could be used with no loss of generality in our approach. In comparison, *TReX* uses statistics within groups of elements defined by structural constraints. More formally, our BM25 formula uses frequency statistics with respect to an element set S rather than using only statistics with respect to entire documents or individual elements. Usually, S is taken to be the set of all elements satisfying the structural constraints of the query. For instance, in the query from Example 1, S contains all the elements in the answer to the XPath expression `//article//sec`.

As tuning parameters we use the same values used in TopX. Thus, we set k_1 to 10.5 and b to 0.75. Note that k_1 controls the non-linear term-frequency effects, and b controls the element-length normalization [10]. In order to answer retrieval queries efficiently, *TReX* uses inverted lists for finding elements that contain the keywords, and summaries for finding elements that comply with the structural constraints. Summaries are discussed in the next section.

3 Structural Summaries

Structural summaries are data structures used for locating specific fragments of the data, such as nodes and subtrees. They group together elements that are indistinguishable with respect to a query or a class of queries in some XML query language. By accessing relevant data directly, summaries help to avoid sequential scans of entire documents during query evaluation. In addition, they can be used to *describe* the instance by keeping record of its structural properties, such as hierarchical relationships, degree of nesting, and label paths. A typical summarization of the XML tree structure is a *labeled tree* that describes its labels and edges in a concise way. In addition, XML tree nodes are partitioned into equivalence classes according to their labels or the label paths they belong to. Each node in the summary tree has one such equivalence class (usually called its *extent* in the literature) associated to it.

The partition can be induced by different criteria. For instance, the *tag summary* clusters together nodes with the same tag. The tag summary has as many extents (equivalence classes) as different tags are in the XML tree. The *incoming summary*, in contrast, partitions nodes based on the label paths from the root to

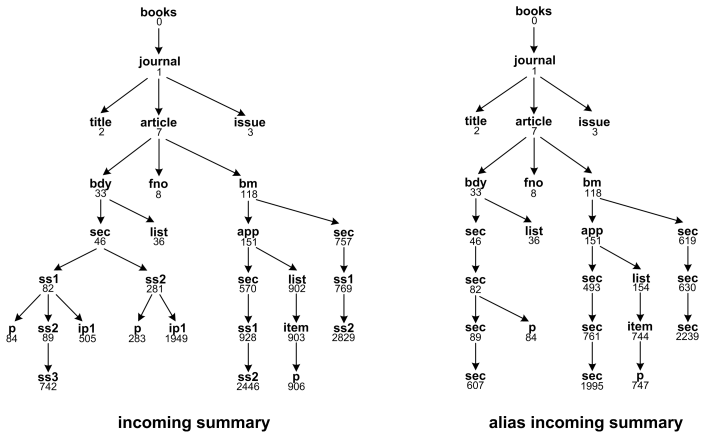


Fig. 1. Fragment of the incoming and alias incoming summary trees for the INEX IEEE collection

the nodes, *i.e.,* the incoming label paths. Thus, nodes with the same incoming label path will belong to the same extent. It is easy to see that the extents of the incoming summary are in fact a refinement of the tag summary extents, because in order for two nodes to have the same incoming label path they also need to have the same label. The left-hand side of Figure 1 shows a fragment of the incoming summary tree for the INEX IEEE collection. (The complete incoming summary with no aliases has 11563 nodes. For the tag summary, the number of nodes is 185. The total size of the alias incoming summary is 7860. The alias tag summary has 145 nodes.) In Figure 1, the numbers below the nodes are the *summary node identifiers*, or sid's for short. For instance, all XML nodes that end with the path `books/journal/article` belong to the same incoming summary extent and, according to our summary in Figure 1, have sid 7. A sid not only identifies a summary node but also includes all XML nodes that belong to the summary node's extent. Note that if two XML nodes have the same sid, then by definition one node cannot encapsulate the other.

In an XML retrieval environment, oftentimes different elements with different tags represent the same type of information. For instance, article sections in the IEEE collection are in some places referred to as `sec` and in other places as `ss1` or `ss2`. Since `sec`, `ss1` and `ss2` are semantically the same. For a summary to reflect that fact, we make use of the *alias mapping* provided by INEX to replace all synonyms by their alias (`sec` in our example). The right-hand side of Figure 1 shows a fragment of the *alias incoming summary* tree for the INEX IEEE collection.

An alias mapping collapses different summary nodes in the non-aliased summary into a single summary node in the aliased summary. This collapse can happen for two different reasons. The first one is that nodes are combined into one because their tags are aliases of the same tag. For instance, nodes with sid's

82 and 281 in the incoming summary of Figure 1 are combined into summary node sid 82 in the alias incoming because tags `ss1` and `ss2` are mapped to (aliased with) `sec`. This type of collapse can happen in both tag and incoming summaries. The second type of collapse is only possible in the incoming summary: two nodes collapse because their ancestors collapse. This is the case of nodes with sid's 84 and 283 on the left-hand side of the figure. When nodes with sid's 82 and 281 were combined into one, the incoming label path to nodes with sid's 84 and 283 became the same and thus the two nodes were also combined into one.

Our system generates an XPath expression for each sid, which computes precisely the set of document nodes in its extent. Attaching an arbitrary XPath expression to each sid gives us the ability to precompute arbitrary path conditions in our summaries. In addition, the use of XPath provides us with a uniform mechanism for creating and manipulating *TReX* summaries.

Since our system uses sid's internally, changing the summary only impacts the sid's used during query evaluation. This provides the flexibility to use different summaries transparently in *TReX*. Any summary proposal in the literature can in fact be used in *TReX*. Examples of such proposals are region inclusion graphs (RIGs) [9], dataguides [12], the T-index family [21], ToXin [22], A(k)-index [16], F&B-Index and F+B-Index [15]. RIGs are examples of tag summaries whereas dataguides, 1-index, ToXin, and A(k)-index are incoming summaries. All these proposals can be expressed in our system using XPath expressions, which gives us the ability mix and match them in our summaries.

3.1 NEXI Evaluation Using Summaries

We now explain how to use structural summaries for evaluating retrieval queries. The evaluation of a *NEXI* retrieval query in *TReX* is done in two phases: translation and retrieval.

In the translation phase, each path p in the query from the root to an `about()` function is translated to a set of sid's and a set of terms. Let E_p be the set of elements in the result of evaluating p on all the documents in the corpus. The set of sid's consists of all the summary nodes whose extent has a non-empty intersection with E_p. The set of terms consists of all the terms that appear in the `about()` function at the end of each path p. For example, consider the query in Example 1 over the INEX IEEE collection, and the incoming summary with aliases shown on the right-hand side of Figure 1. Then, the set of sid's for the path `//article//sec` is $\{46, 82, 89, 493, 607, 619, 630, 761, 1995, 2239\}$. The set of terms is $\{\text{inverted}, \text{list}\}$. For the path `//article` that also leads to an `about()` function, the set of sid's is $\{7\}$ and the set of terms is $\{\text{XML}, \text{retrieval}\}$.

In the retrieval phase, elements are retrieved according to the sets of sid's and terms generated in the translation phase. For a set of sid's $[sid_1, \ldots, sid_m]$ and a set of terms $[t_1, \ldots, t_n]$, the system retrieves the elements that *(1)* are in the extent of a node with sid in $sid_1, \ldots, sid_m$, and *(2)* contain at least one of the terms $t_1, \ldots, t_n$. For each such element e, term and element frequencies are computed and a BM25 score $score_s(e \mid t_1, \ldots, t_n)$ is calculated, where S

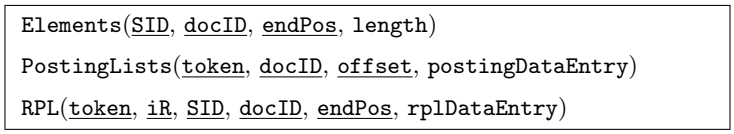


Fig. 2. The schemes of the tables *TReX* stores.

is the extent in which e is a member. The following section discusses how the algorithms of the retrieval phase were implemented in *TReX*.

4 Exhaustive Retrieval Algorithm

In this section, we describe the exhaustive retrieval algorithm (*ERA*) for the retrieval phase of query evaluation. As explained in Section 3.1, the input to *ERA* consists of a list of sid's and a list of terms. An element of a document in the corpus is considered *relevant*, if *(1)* it is in the extent of one of the given sid's and *(2)* it contains at least one of the given terms. *ERA* finds all the relevant elements. In addition, for each relevant element e and for each term t among the given terms, *ERA* computes the frequency of t in e, (*i.e.,* the number of times that t appears in e). These term frequencies are the basis for ranking the elements of the result, as was discussed in Section 2. Note that *ERA* can be used not only with BM25 but also with any other ranking method that is based on term frequency.

For evaluating queries, *ERA* uses a structural summary of the corpus and inverted lists. An inverted list stores all the positions where each term appears. Positions are represented in *TReX* as pairs of a document identifier and an offset from the beginning of the document. Summaries and inverted lists are stored as indexed relational tables. The following section describes these tables, and in Section 4.2 we present *ERA*.

4.1 Data Structures

In *TReX*, the structural summary and the inverted lists are stored in two indexed tables named `Elements` and `PostingLists`. The schemes of these tables are shown in Figure 2. In the figure, keys are underlined. For each table, an index provides ordered sequential access to the tuples according to the keys.

The `Elements` table contains an entry for each element in the corpus. `SID` is the summary id of the element. The field `docID` holds the identifier of the document in which the element appears. The `endPos` is the position in the document where the element ends, and `length` is the length of the element. Note that we can compute the start position of each element by subtracting the length from the end position.

The `PostingLists` table is actually the inverted lists. For each term, all the positions where this term appears are stored in the table. The position of the

term is represented by the identifier of the document in which the term appears and an offset from the beginning of this document. The `token` field is the token (*i.e.,* term) that the entry represents. In each tuple, the `postingDataEntry` is a list of the form $doc_1, o_1^1, \ldots, o_{i_1}^1, doc_2, o_1^2, \ldots, o_{i_2}^2, \ldots, doc_k, o_1^k, \ldots, o_{i_k}^k$ where $doc_1, \ldots, doc_k$ is a sorted list of document identifiers, and each $o_1^j, \ldots, o_{i_j}^j$ is a sorted list of offsets indicating the positions where the token appears in the document doc_j. The posting list may become too long for storing it in a single tuple. So it may be divided and stored across several tuples. In order to access the parts of the posting list in order of position, the fields `docID` and `offset` in `postingDataEntry` are part of the key.

For technical reasons, we also add a *maximal* dummy position denoted *m-pos* to the end of the last `postingDataEntry` list of each term. The position *m-pos* is maximal in the sense that no real position can exceed it. This is done to detect the end of each posting list.

4.2 The Exhaustive Algorithm

We now show how *ERA* computes a query result from the data in the `Elements` and `PostingLists` tables. The main code is presented in Figure 3. Before we explain the code, we describe the iterators used in *ERA*. There are two principle iterators; one for the `Elements` table and the second for the `PostingLists` table. The first iterator searches over the index of `Elements`. For a sid s, let iterator I_s return all the positions of relevant elements in s in ascending order of (`docID`, `endPos`). The function call I_s.*firstElement*() returns the first tuple in `Elements` whose sid is equal to s. The function call I_s.*nextElementAfter*(p) returns the element with the lowest position greater than p in extent s where p is a tuple of the form (`docID`, `endPos`). If no element is found then a dummy element is returned—an element with end position equal to *m-pos* and length equal to zero. The second iterator searches over the index of `PostingLists`. For a given term t, an iterator I_t over the posting list of t is created. It contains a single function I_t.*nextPosition*() that successively returns the next position in the posting list of t.

We now explain the code of *ERA* given in Figure 3. The input to the algorithm consists of a list of sid's $sid_1, \ldots, sid_m$ and a list of terms $t_1, \ldots, t_n$. The initialization of the algorithm involves creating variables for results and the necessary iterators. Lines 1 and 2 creates an empty list L to store the results of the computation and an array C of size $m \times n$ to keep intermediate count values of appearances of terms in elements. The purpose of C is to record for m different elements how many times each term among $t_1, \ldots, t_n$ has been seen in these elements. For each sid and term, iterators over `Elements` and `PostingLists` respectively, are constructed in lines 3–8 and the initial values from these iterators are stored in vectors e_i and pos_j, respectively.

After the initialization, the algorithm iterates over all the positions where one of the given terms appears. In each iteration, the lowest position not handled so far is being considered. We denote this position by pos_x and the term that it

$\underline{ERA((sid_1, \ldots, sid_m), (t_1, \ldots, t_n))}$

Input: A list of sid's and a list of terms
Output: The relevant elements with their term frequencies

```
 1: let L be a new empty list
 2: let C[m][n] be an array of size m × n having 0 in all the cells
 3: for i = 1 to m do
 4:     create a new iterator I_{sid_i} over elements in the extent of sid_i
 5:     e_i ← I_{sid_i}.firstElement()
 6: for j = 1 to n do
 7:     create a new iterator I_{t_j} over the positions of t_j
 8:     pos_j ← I_{t_j}.nextPosition(t_j)
 9: repeat
10:     let x be the index for which pos_x = min{pos_1, ..., pos_n}, and let t_x be the term
        that starts in position pos_x
11:     for i = 1 to m do
12:         if pos_x < start(e_i) then
13:             {do nothing}
14:         else if start(e_i) < pos_x < end(e_i) then
15:             C[i][x] ← C[i][x] + 1
16:         else if end(e_i) < pos_x then
17:             if there is a non-zero cell in the row C[i][1, ..., n] then
18:                 create a new list tf_{e_i} from the n values C[i][1, ..., n]
19:                 add (e_i, tf_{e_i}) to L
20:                 reset all the cells C[i][1, ..., n] to 0
21:             e_i ← I_{sid_i}.nextElementAfter(pos_x)
22:             if start(e_i) < pos_x < end(e_i) then
23:                 C[i][x] ← C[i][x] + 1
24:     pos_x ← I_{t_x}.nextPosition()
25: until for all the terms, the maximal position m-pos has been reached
26: return L
```

Fig. 3. Retrieving the relevant elements.

refers to by t_x. For the term t_x and each one of the elements that are currently being processed, the algorithm checks whether these elements contain t_x and updates C accordingly. More precisely, when an element e_i is being processed, it has three possible relationships with t_x, which we explain next.

If the element e_i starts after pos_x, then t_x is not contained in e_i and the counts in C should not be changed. Yet, at this point, term appearances in positions greater than pos_x may be inside e_i. Thus, e_i still needs to be processed. In this case, no action is being done (lines 12–13). If pos_x is between the start position of e_i and the end position of e_i then we encountered an appearance of t_x inside e_i. In this case, the counting in C is updated (lines 14–15).

If the element e_i ends before pos_x, then there is no need to change C. Furthermore, since all the following appearances of terms will be in a position greater than pos_x, at this point in the run, the counting of frequencies for e_i is complete and we can replace e_i with the next element from the extent of sid_i. If at least one of the term frequencies of e_i is greater than zero, then we add e_i and its frequencies to the list L (lines 17–20). We then replace e_i with the next element in the extent of sid_i (line 21) and start the counting for this element. Note that the term being processed can be inside the new element and in this case we need to immediately update the counting for this new element (lines 22–23).

When the dummy maximal position has been reached for all terms, the computation is complete and L can be returned. *TReX* implements *ERA* using

iterators so that relevant elements can be provided as soon as the computation of their term frequencies is complete. We do not provide the details in this paper. In post-processing, we compute the BM25 scores for the retrieved elements and sort them by their respective scores.

4.3 Relevance Posting Lists

ERA finds the relevant elements and, initializes them with their term frequencies, and sorts them by their end position. After computing the BM25 score of each element and sorting the elements by these scores, the result is stored because these results can be used to efficiently evaluate the query as a top-k query. *TReX* stores these results as *relevance posting lists* (*RPLs*) of the terms. An RPL of a term t is a list of elements that contain t, with each element's relevance score and sid. Elements in an RPL are sorted according to their relevance, in descending order. Rather than physically storing and maintaining many different lists, in *TReX*, all RPLs are stored in a single relation named `RPL`. The schema of this relation is shown in Figure 2. Each tuple in the `RPL` relation contains part of the RPL of some term t. The term t is stored in the `token` field, and the RPL (or a part of it) is stored in the `rplDataEntry` field. The field `rplDataEntry` holds a list of 5-tuples, where each 5-tuple identifies an element and consists of *(1)* a relevance score, *(2)* an sid, *(3)* a document identifier, *(4)* an offset to end position, and *(5)* a length. The elements in `rplDataEntry` are sorted in a decreasing order according to their relevance score. For each 5-tuple, the combination of sid, document identifier and offset-to-end are used as unique identifiers for elements. The attributes `iR`, `SID`, `docID` and `endPos` in `RPL` contain the values of the first element in `rplDataEntry` for ordering divided lists.

In *TReX*, given a list $sid_1, \ldots, sid_m$ of sid's and a list $t_1, \ldots, t_n$ of terms, RPLs can be used to efficiently compute top-k answers. Let t be one of the terms $t_1, \ldots, t_n$. The top-k relevant elements with respect to$t$ and $sid_1, \ldots, sid_m$ can be easily retrieved from the RPL of t by iterating over this RPL and selecting the top elements whose sid is among $sid_1, \ldots, sid_m$. Note that the elements are provided sorted by their rank. We can then use threshold algorithm (TA), similar to the one used in TopX [24], in order to combine for each element its scores in the n RPLs, and return the top-k answers. Note that this algorithm is a version of the TA algorithm proved by Fagin *et al.* [11] which is instance optimal in terms of the number of readings from the lists.

5 Experimental Results

We experimented with *TReX* in order to measure the efficiency and effectiveness of our retrieval methods. Two other goals of our experiments were to investigate the influence of using different summaries on the system's performance and to compare the running time of *ERA* against TA. We implemented *TReX* in Java and used Berkeley DB (BDB) for the indexed tables. Our initial experiments were conducted over the IEEE collection provided in the INEX 2005 benchmark.

This collection contains 16819 XML documents, and it has a size of 0.76GB. For the IEEE collection, the sizes of the tables `Elements` and `PostingLists`, stored in BDB, were 1.52GB and 8.05GB, respectively. Follow up experiments were conducted on the Wikipedia collection which contains approximately 645,719 documents and has a size of 5.01GB. The follow up experiments used the same basic configuration as was used for the IEEE collection.

Table 1. NEXI Queries and Translations for IEEE.

Query ID	*NEXI* Query
203	`sec[about(., code signing verification)]`
223	`article[about(.//sec, wireless ATM multimedia)]`
233	`article[about (.//bdy, synthesizers) and about (.//bdy, music)]`
236	`article[about(., machine translation approaches -programming)]`
260	`bdy//*[about(., model checking state space explosion)]`

Query ID	Tag sid's	Incoming sid's	Keywords
203	6, 40	7, 46, 82, 89, 493, 607, 619, 630, 761, 1995, 2239	code, signing, verification
223	6, 40	7, 46, 82, 89, 493, 607, 619, 630, 761, 1995, 2239	wireless, ATM, multimedia
233	6,32	7,33	synthesizers, music
236	6	7	machine, translation, approaches
260	6, 32	7, 33	model, checking, state, space, explosion

We tested *TReX* on many INEX queries; however, we report here only the detailed results of five arbitrary queries from IEEE that seemed to us as representing the typical behavior of all the other queries. Similarly, the follow up results from five arbitrary queries from Wikipedia show that performance with a larger corpus was comparable to IEEE results. Table 1 shows the queries we chose and the translation of the IEEE queries for both the tag summary and incoming summary. Table 2 shows the queries we chose and the number of sid's used in the query translations for the incoming summary.

Table 2. NEXI Queries and Number of sid's in Translations for Wikipedia

Query ID	*NEXI* Query	# of sid's
291	`article//figure[about(., Olympian god goddess)]`	1388
292	`article//figure[about(., Renaissance painting Italian Flemish -French -German)]`	1388
346	`article[about(.,+unrealscript language api tutorial)]`	4
356	`article[about(.,natural language processing) and about(.,information retrieval)]`	4
388	`article[about(.,rhinoplasty)]`	4

The Wikipedia results in Table 5 were generated using incoming summaries with alias. The structure of the summary tree for Wikipedia is significantly

Table 3. Average Evaluation Time (in seconds) *ERA* Using Incoming and Tag Summaries for IEEE.

Query ID	Tag Summary	Incoming Summary	Efficiency Improvement
203	4873	1651	66%
233	1991	696	65%
236	5643	1812	68%
260	8860	1640	81%

Table 4. Average Evaluation Time (in seconds) TA Using Incoming Summary for IEEE.

Query ID	top10	top50	top100	top500	top1000	top1500
203	28	61	93	227	312	486
233	0.59	0.94	0.98	1	0.77	0.73
236	4	16	21	41	53	60
260	14	59	92	237	359	460

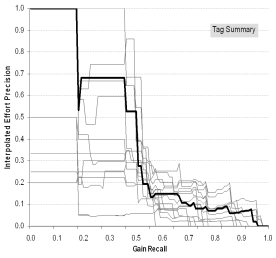
larger and more complex than that of the IEEE corpus. The Wikipedia contains about 6 times more sid's than IEEE. The evaluation times for Wikipedia were in the same scale of magnitude as IEEE. The IEEE topics were structurally constrained to article bodies and article sections. Wikipedia queries 291 and 292 were constrained to figures in articles. Wikipedia queries 346, 356 and 388 were structurally constrained to articles. From these results, we conjecture that the factors in determining the running time of queries are the number of sid's considered, the size of the sid summary extents, and, most importantly, the number of matching tokens in the corpus.

Although we evaluated our queries on both the IEEE collection and Wikipedia, we measure the effectiveness of our retrieval techniques only on the IEEE collection. The results are presented in Tables 3 and 4. We compared our results to the results of other INEX participants. This is shown below in Figure 4. We ran the comparisons using the INEX Evaluation Package EvalJ[2]. In this comparison, recall and precision of query results are computed based on ranking performed by humans.

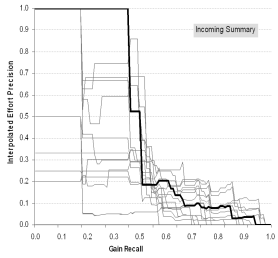
Figure 4 shows the comparative effectiveness of two representative queries, Query 203 and Query 223 (listed in Table 1), using the tag summary and the incoming summary. The results of *TReX* are depicted with a bold line whereas the results of other INEX participants are depicted with light gray lines. Intuitively, each line shows the precision gained, as a function of the recall, for a single system. That is, a line of a system S going through a point (r, p) means that for a

Table 5. Evaluation Time (in seconds) *ERA* Using Incoming for Wikipedia.

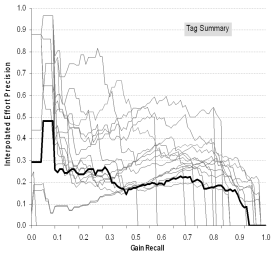
Query ID	Incoming Summary
291	1953
292	3435
346	1092
356	1283
388	637



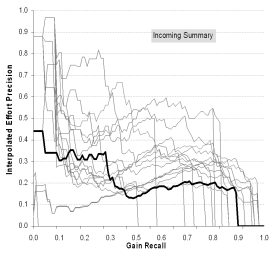
(a) Query 203 - Tag Summary



(b) Query 203 - Incoming Summary



(c) Query 223 - Tag Summary



(d) Query 223 - Incoming Summary

Fig. 4. Comparative effectiveness of *TReX* using EvalJ among other INEX 2005 participants.

given k the top-k answers have a recall of r, and the precision of these k answers is p. The graphs in Figure 4 show that the incoming summary provides better results than the tag summary; however, the superiority of the incoming summary is not always the case. Note that, for Query 203, when using the incoming summary, 50% of the elements a human would include in the answer were given the highest scores by *TReX*, which means that they could be retrieved with 100% precision. Our tests suggest that the effectiveness of *TReX* is comparable to, and in many cases better than, the effectiveness of other systems that participated in INEX.

An important conclusion from our experiments is that summaries have a major influence on the efficiency and effectiveness of the system. Specifically, *TReX* had performed better with incoming summary than with tag summary. One explanation of this is that the tag summary does not take into account the ancestor-descendant relationship among elements, and thus, the partition it provides for the elements is coarser than the partition provided by the incoming summary. This means that every sid represents more elements, and so, more elements need to be processed by *ERA*. Also, using summaries causes query results to be less accurate because the structural constraints are evaluated in a flexible way that stems from the type of summary employed. These results are

promising but not definitive. In the future, we hope to address this issue with more broad-ranging experimental results. We leave the question of how to choose an appropriate summary for future work.

6 Conclusion

In this paper we presented *TReX*—a system for efficient XML retrieval using summaries. The main contribution of our work is showing how to utilize summaries for a vague interpretation of structural constraints: either when all the answers to a query must be returned or when only the top-k answers are needed. We tested our retrieval algorithm on data and queries from INEX. The tests show that our retrieval method is efficient and effective. Our results provide a new and general perspective to structural evaluation in INEX. The flexibility and efficiency of the approach is coupled with a general framework so XML summaries can be easily incorporated into any XML retrieval system. Future work includes a study of the potential of using summaries for answering queries under a strict interpretation of the structural constraints. It also includes a study of the relationship between exhaustive retrieval and top-k query answering.

References

1. INEX: Initiative for the evaluation of XML retrieval. `http://inex.is.informatik.uni- duisburg.de:2005`, 2005.
2. EvalJ: INEX evaluation package. `http://evalj.sourceforge.net`, 2006.
3. S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying structured text in an XML databases. In *Proc. SIGMOD Conf.*, pages 4–15, 2003.
4. S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: a full-text search extension to XQuery. In *Proc. WWW Conf.*, pages 583–594, 2004.
5. S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: flexible structure and full-text querying for XML. In *Proc. SIGMOD Conf.*, pages 83–94, 2004.
6. A. Barta, M. P. Consens, and A. O. Mendelzon. Benefits of path summaries in an xml query optimizer supporting multiple access methods. In *Proc. VLDB Conf.*, pages 133–144, 2005.
7. J. Clark and S. DeRose. XML Path Language (XPath) version 1.0. `http://www.w3.org/TR/xpath`, 1999.
8. S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A semantic search engine for XML. In *Proc. VLDB Conf.*, pages 45–56, 2003.
9. M. P. Consens and T. Milo. Optimizing queries on files. In *Proc. SIGMOD Conf.*, pages 301–312, 1994.
10. R. et al. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proc. SIGIR Conf.*, pages 232–241, 1994.
11. Fagin and et al. Optimal aggregation algorithms for middleware. In *Proc. PODS Conf.*, pages 102–113, 2001.
12. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB Conf.*, pages 436–445, 1997.
13. L. Guo and et al. XRANK: Ranked keyword search over XML documents. In *Proc. SIGMOD Conf.*, pages 16–27, 2003.

14. V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *Proc. ICDE Conf.*, pages 367–378, 2003.
15. Kaushik and et al. Covering indexes for branching path queries. In *Proc. SIGMOD Conf.*, pages 133–144, 2002.
16. Kaushik and et al. Exploiting local similarity for indexing paths in graph-structured data. In *Proc. ICDE Conf.*, pages 129–140, 2002.
17. Kaushik and et al. On the integration of structure indexes and inverted lists. In *Proc. SIGMOD Conf.*, pages 779–790, 2004.
18. W. Lu, S. E. Robertson, and A. MacFarlane. Field-weighted xml retrieval based on bm25. In *Proc. INEX Workshop*, pages 161–171, 2006.
19. S. Malik and et al. Overview of INEX 2005. In *Proc. INEX Workshop*, 2005.
20. A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in XML. In *Proc. ICDE Conf.*, pages 162–173, 2005.
21. T. Milo and D. Suciu. Index structures for path expressions. In *Proc. ICDT Conf.*, pages 277–295, 1999.
22. F. Rizzolo and A. O. Mendelzon. Indexing XML data with ToXin. In *Proc. WebDB Workshop*, pages 49–54, 2001.
23. T. Schlieder and H. Meuss. Querying and ranking XML documents. *Journal of the American Society for Information, Science and Technology*, 53(6):489–503, 2002.
24. M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for TopX search. In *Proc. VLDB Conf.*, pages 625–636, 2005.
25. A. Trotman and B. Sigurbjornsson. Narrowed extended XPath I (NEXI). In *Proc. INEX Workshop*, pages 16–39, 2004.