Self Managing Top-k (Summary, Keyword) Indexes in XML Retrieval

Mariano P. Consens Xin Gu Yaron Kanza Flavio Rizzolo
University of Toronto
{consens, xgu, yaron, flavio}@cs.toronto.edu

Abstract

Retrieval queries that combine structural constraints with keyword search represent a significant challenge to XML data management systems. Queries are expected to be answered as efficiently and effectively as in traditional keyword search, while satisfying additional constraints. Several XML-retrieval systems support answering queries exhaustively by storing both structural indexes and a keyword index. Other systems answer top-k queries efficiently by constructing indexes in which keyword scores, for some structural elements, are stored in relevance order, enabling approaches such as the threshold algorithm (TA).

In this paper we describe TReX, an XML retrieval system that can exploit multiple structural summaries *(including newly defined ones). TReX can also self-manage small, redundant, indexes to speed up the evaluation of workloads of top-k queries. The redundant indexes are maintained to enable TReX to select an evaluation strategies among three (and potentially more) retrieval methods. We provide experimental evidence that using several strategies improves the efficiency of query evaluation, since none of the retrieval methods outperforms the others in all cases.*

1. Introduction

In XML documents, structure and content are inseparable. Thus, several languages for XML retrieval, which have been recently developed, allow specifying structural constraints in addition to keywords. An example of such language is NEXI [18], which has been developed in the context of INEX, an initiative that provides a forum for researchers to demonstrate retrieval capabilities on collections of XML documents.

NEXI. In NEXI (*Narrowed Extended XPath I*), queries combine keywords with structural constraints. A query answer is a ranked list of XML elements that (1) contain at least one of the keywords, and (2) satisfy the constraints. Each implementation of NEXI has its own ranking criteria, which generally use well-established IR techniques, for content scoring, in addition to extensions that account for the structural constraints.

NEXI is built upon XPath [1]. On one hand, it narrows XPath by excluding some function symbols and some axes of XPath. On the other hand, it extends XPath by adding to it the function `about()`. The `about()` function is used for filtering elements based on their relevance to a specified list of keywords.

Example 1.1 Consider the following NEXI retrieval query: `//article[about(., XML)]//sec[about(., query evaluation)]`. This query specifies a search, for sections that deal with "query evaluation" and appear in an article that is about "XML". The `about()` function specifies the keywords for the search.

There are two possible interpretations to NEXI queries. Under a *strict interpretation*, the structural constraints should be satisfied precisely. Under a *vague interpretation*, the structural constraints are relaxed and need to be only loosely satisfied. The answer to a retrieval query consists of elements that satisfy the structural constraints—strictly or vaguely according to the interpretation—and contain at least one of the specified keywords. The elements in the answer are ranked according to their relevance to the search. For instance, under a strict interpretation, the answer to the query of Example 1.1 consists of `sec` elements that are descendants of `article` elements, *i.e.*, elements that are in the answer to the XPath expression `//article//sec`. The `sec` elements in the answer should be ranked according to their relevance to the keywords "query" and "evaluation", and the relevance of their ancestor `article` elements to the keyword "XML". Under a vague interpretation, the answer to the query is similar except that the `article` and `sec` tags can be replaced by any other tag names, presumably having the same meaning.

There are two main challenges when implementing a retrieval language such as NEXI. First, ranking answers as close as possible to a ranking performed by a human. Providing such ranking is beyond the scope of this paper. The second challenge, which is addressed in this paper, is computing retrieval queries efficiently.

The TReX System. In this paper we describe *TReX*—a system for efficient evaluation of vague NEXI queries. *TReX* can either find all the answer to a query or only the top-k answers. It employs three retrieval methods, namely *exhaustive*, *TA* and *merge*, in order to choose the best one according to the query. Each method requires its own indexes. But, when disk space is limited, only some of the indexes should be created and stored. One of the novelties of *TReX* is self-managing the creation of indexes used by the TA and merge algorithms, for efficient evaluation of queries according to a given *workload*.

Related Work. Several proposals in the literature extend the traditional keyword-style retrieval to the XML model [5, 8, 9]. Vague structural conditions were introduced in [16] and complemented with full-text conditions in [2, 3]. A query algebra for IR style processing of XML data was introduced in [4]. Efficient evaluation of keyword top-k queries was considered in XRANK [8]. Other recent proposals for XML ranked retrieval include [11] and [13]. The former uses dataguides and TA-style top-k algorithms [6], but differs from our work in that their experiments are limited to DB-like queries rather than XML retrieval ones. In contrast, [13] focuses on efficient evaluation of approximate structural matches without considering keyword search.

Contributions. The main contributions of this paper are as follows.

- A novel use of summaries for efficient XML retrieval is presented (Section 2).
- The approach of materializing top-k indexes for efficient evaluation of top-k queries using different strategies investigated (Section 3).
- It is shown how to self-manage materialized redundant (element, tag, score) indexes for increasing the efficiency of query evaluation while minimizing the use of disk space (Section 4).
- Experiments are provided to show the benefits of using different strategies for evaluating different queries (Section 5).

2. Summaries and Retrieval Indexes

Two basic data structures to support efficient XML retrieval in *TReX* are *inverted lists* and *structural summaries*. Inverted lists support retrieval of elements that contain specific keywords, while summaries support retrieving elements that satisfy certain structural conditions. In this section we shortly describe these data structures.

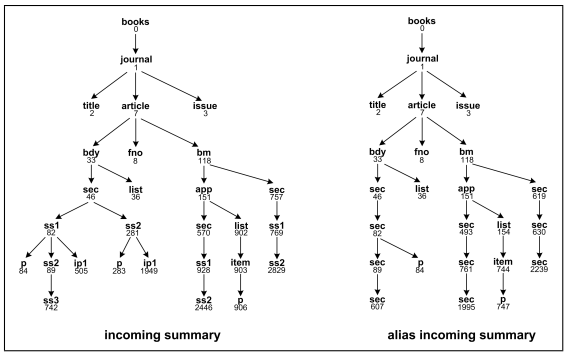

Figure 1. Summary fragments for INEX IEEE

2.1 Structural Summaries

Structural summaries are data structures used for locating specific fragments of the data, such as nodes and subtrees, according to structural constraints. 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 they help to avoid sequential scans of entire documents during query evaluation. In addition, they can be used to *describe* an instance by keeping record of its structural properties, such as hierarchical relationships, degree of nesting, and label paths.

A typical summarization of the structure of an XML document is a *labeled tree* that describes, in a concise way, the labels and edges of the document. Each node in the summary tree has one such equivalence class (usually called *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 (same label). The tag summary has as many extents as different tags are in the XML document. The *incoming summary*, in contrast, partitions nodes based on the label paths from the root to them, *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: 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 numbers below the nodes are *summary node identifiers*, or sids for short. For instance, all the XML nodes, in the example, that are reachable from the root by the path `books/journal/article` belong to the same incoming-summary extent and have sid 7.

In an XML retrieval environment, oftentimes different elements with different tags represent the same type of information. For instance, article sections in the IEEE col-

lection are in some places referred to as `sec` and in some others as `ss1` or `ss2`. Since `sec`, `ss1` and `ss2` are semantically the same, we would like our summary to reflect that fact. Therefore, 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. For the IEEE collection, the complete incoming summary with no aliases has 11563 nodes. For the tags summary, the number of modes is 185. The total size of the alias incoming summary is 7860. The alias tag summary has 145 nodes.

For efficiency, *TReX* uses only summaries in which there are no two XML elements in the same extent where one encapsulates the other. That is, every pair of ancestor-descendant elements have different sids. Using alias mappings this can be easily guaranteed.

TReX uses the alias incoming summary where the extents are described using XPath expressions. Most of the summaries proposed in the literature can in fact be described using XPath expressions and be used in *TReX*. Examples of such proposals are dataguides [7], the T-index family [14], ToXin [15], A(k)-index [12], F&B-Index and F+B-Index [10].

2.2 Indexes in TReX

In *TReX*, a structural summary and inverted lists are stored in two indexed tables named `Elements` and `PostingLists`. Two additional indexes that store triplets of term, element and ranking score are *relevance posting lists* (`RPLs`) and *element-relevance posting lists* (`ERPLs`). The schemes of these tables are as follows.

`Elements(SID, docID, endPos, length)`

`PostingLists(token, docID, offset, postingDataEntry)`

`RPLs(token, iR, SID, docID, endPos, rplDataEntry)`

`ERPLs(token, SID, docID, endPos, iR, erplDataEntry)`

Primary keys are underlined. For each table, an index on the primary key provides a sequential access to the tuples.

The `Elements` table contains an entry for each element in the corpus. `SID` is the summary id of the element. `docID` is the id 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.

The `PostingLists` table holds the inverted lists. For each term, stored in the `token` field, the table holds in the `postingDataEntry` field all the positions where the term appears. A position is represented by a pair of document identifier and an offset from the beginning of the document. Since the posting list might be too long for storing it in a single tuple, it is divided and stored in several tuples whenever needed. In order to access the fragments of a posting list that has been divided, according to position order, the first position (`docID` and `offset` fields) in each fragment is part of the key.

For technical reasons, we 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 in order to facilitate the handling of the case where the end of the posting list is reached.

A relevance posting list (RPL) of a term t stores all the elements that contain t with their relevance scores. *TReX* stores RPLs in the tables `RPLs` and `ERPLs`. These tables differ in the order by which elements are kept. In `RPLs` elements are sorted in descending order of relevance, while in `ERPLs` elements are sorted by position.

Each tuple, in either `RPLs` or `ERPLs`, holds a term and some (not necessarily all) of the elements that contain the term. 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.

TReX uses RPLs for computing the top-k answers to a given query. Suppose the translation of a given query is the lists $sid_1, \ldots, sid_m$ of sids and the list $t_1, \ldots, t_n$ of terms. The top-k answers of the query can be computed by selecting from the RPLs of $t_1, \ldots, t_n$ the elements in the extents of $sid_1, \ldots, sid_m$ and merging the result lists, *e.g.*, using a merge algorithm or a threshold algorithm.

3. Retrieval Strategies

In this section we describe the evaluation of queries in *TReX* and present the retrieval methods that are used.

3.1 Query Evaluation Using Summaries

Evaluation of a NEXI 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 sids 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. Then, the set of sids consists of all the summary nodes whose extent has a non-empty intersection with E_p, whereas the set of terms consists of all the terms that appear in the `about()` function at the end of p. For example, consider the query in Example 1.1 over the INEX IEEE collection, and the incoming summary with aliases shown on the right-hand side of Figure 1. Then, the set of sids for the path `//article//sec` is {46, 82, 89, 493, 607, 619, 630,761, 1995, 2239}. The set of terms is {query, evaluation}. For the path `//article` that also leads to an `about()` function, the set of sids is {7} and the set of terms is {XML}.

In the retrieval phase, elements are retrieved according to the sets of sids and terms generated in the translation phase. For each set $sid_1, \ldots, sid_m$ of sids and set $t_1, \ldots, t_n$ of terms, the system retrieves the elements that *(1)* are in the extent of a node with sid among $sid_1, \ldots, sid_m$, and *(2)* contain at least one of the terms $t_1, \ldots, t_n$. Retrieval strategies are discussed next.

3.2 Exhaustive Retrieval Algorithm

The exhaustive retrieval algorithm (*ERA*) evaluates queries using the `Elements` and `PostingLists` tables. The main code is presented in Figure 2. *ERA* computes queries using two types of iterators over the indexes. For each given sid, *ERA* creates an iterator that returns all the elements in the extent of this sid. The elements are retrieved from the `Elements` table. Each element is identified by the position where it ends (the values in the fields `docID`, `endPos` of `Elements`). Suppose that an iterator I_s is created for a given sid s. Then, the function call I_s.*firstElement*() returns the first tuple in `Elements` whose sid is equal to s. (Remember there is an index on the key that provides a sequential access to the tuples of `Elements`.) Given a position p, the function call I_s.*nextElementAfter*(p) returns the element with the lowest position greater than p among the elements in the extent of s. 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. This function is implemented as a search over the index of `Elements`. Iterators over the postings (positions) of a given term are also being used. For each given term t, an iterator I_t over the posting list of t is created. Function I_t.*nextPosition*() returns the next position in the posting list of t.

Using *ERA*, *TReX* can compute all the answers to a given query. *TReX* also uses *ERA* for generating or extending the `RPLs` and `ERPLs` tables.

3.3 Threshold Algorithm

One of the methods implemented in *TReX* is the threshold algorithm (*TA*) [6] in a version similar to the implementation that has been used in TopX [17]. Essentially, *TA* generates, for each given term t of the query, a term iterator I_t over the `RPLs` table. The iterator I_t returns the elements that contain t sorted according to their score, where elements that do not have an sid among the sids provided in the query are skipped. For each element, *TA* combines the scores from the iterators, and eventually, for some given k, returns the k elements with the top score combinations.

ERA$((sid_1, \ldots, sid_m), (t_1, \ldots, t_n))$

Input: A list of sids 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 $m \times n$ 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: $e_i \leftarrow I_{sid_i}$.*firstElement*()
6: **end for**
7: **for** $j = 1$ to n **do**
8: create a new iterator I_{t_j} over the positions of t_j
9: $pos_j \leftarrow I_{t_j}$.*nextPosition*(t_j)
10: **end for**
11: **repeat**
12: let x be the index for which $pos_x = \min\{pos_1, \ldots, pos_n\}$, and let t_x be the term that starts in position pos_x
13: **for** $i = 1$ to m **do**
14: **if** $pos_x < start(e_i)$ **then**
15: {*do nothing*}
16: **else if** $start(e_i) < pos_x < end(e_i)$ **then**
17: $C[i][x] \leftarrow C[i][x] + 1$
18: **else if** $end(e_i) < pos_x$ **then**
19: **if** there is a non-zero cell in the row $C[i][1, \ldots, n]$ **then**
20: create a new list tf_{e_i} from the n values $C[i][1, \ldots, n]$
21: add (e_i, tf_{e_i}) to L
22: reset all the cells $C[i][1, \ldots, n]$ to 0
23: **end if**
24: $e_i \leftarrow I_{sid_i}$.*nextElementAfter*(pos_x)
25: **if** $start(e_i) < pos_x < end(e_i)$ **then**
26: $C[i][x] \leftarrow C[i][x] + 1$
27: **end if**
28: **end if**
29: **end for**
30: $pos_x \leftarrow I_{t_x}$.*nextPosition*()
31: **until** for all the terms, the maximal position *m-pos* has been reached
32: **return** L

Figure 2. The ERA algorithm.

3.4 Merge Algorithm

The *Merge* algorithm is presented in Figure 3. While *TA* iterates over RPLs, *Merge* evaluates queries using ERPLs. The algorithm generates iterators for the given terms using the table `ERPLs`. It then combines the scores from the iterators and eventually sorts the elements according to the combined scores.

4. Self-Managing Retrieval Indexes

TReX evaluates a given query by choosing a method from the three evaluation methods that were presented in Section 3. *ERA* can be used for evaluating any given query, provided that the inverted lists and the structural summary exist; however, as we show in Section 5, for many queries *Merge* and *TA* are much more efficient than *ERA*. For using *Merge*, the system should maintain `ERPLs`, and for using *TA*, the system should store `RPLs`.

Theoretically, a system can store for each pair of term and sid both an RPL and an ERPL. Then, given a lists of terms $t_1, \ldots, t_m$ and a list of sids $sid_1, \ldots, sid_n$, evaluation can be done in two steps. First, for each given term

Merge$(L_1, \ldots, L_n)$

Input: ERPLs $L_1, \ldots, L_n$ of the terms $t_1, \ldots, t_n$;
Output: A list V of merged elements sorted by score.

```
1: V ← ∅
2: for i = 1 to n do
3:    create an iterator I_i over the elements of L_i
4:    c_i ← I_i.firstElement()
5: end for
6: repeat
7:    m ← The minimal position among the positions of the elements in
      c_1, ..., c_n
8:    let e be the element in position m
9:    score ← 0
10:   for i = 1 to n do
11:      if the position of the element in c_i is m then
12:         add the score of c_i to score
13:         if I_i.hasNext() then
14:            c_i ← I_i.next()
15:         else
16:            c_i ← dummy element
17:         end if
18:      end if
19:   end for
20:   add (e, score) to V
21: until for 1 ≤ i ≤ n, c_i is the dummy element , i.e., for all the iterators I_i
    the end has been reached
22: sort V using QuickSort
23: return V
```

Figure 3. The *Merge* Algorithm.

t_i, scored elements from the n RPLs in ERPLs that correspond to t_i and s_j (for $j = 1, \ldots, n$) are merged. Secondly, the lists that were generated in the first step are merged using *Merge*. A similar computation can be done using RPLs and *TA*. If the two computations are being done in parallel, the system can return the answer from the computation that finishes first.

In practice, storing an RPL and an ERPL for each sid, term pair requires a lot of disk space. Thus, a system should store only the lists that contribute the most to the efficiency of handling a given *workload*. In the rest of this section we will consider the problem of which indexes to store.

We assume that there is a set of typical queries that are frequently being posed to the system. A workload expresses these queries and their frequencies.

Definition 4.1 (Workload) *A* workload *is a list of top-k retrieval queries $Q_1, \ldots, Q_l$, where each query Q_i is associated with a frequency $0 < f_i \leq 1$, such that $\Sigma_{i=1}^{l} f_i = 1$.*

Given a workload, we need to decide for each query whether to generate and store RPLs or ERPLs for the terms and sids of this query. This depends on the speedup gained by the additional indexes and the available disk space. Consider a top-k query Q_i whose translation is the lists $t_1, \ldots, t_m$ and $sid_1, \ldots, sid_n$. Let T_e be the time it takes to evaluate Q_i using *ERA*. Let T_m and T_{ta} be the times for computing Q_i using *Merge* and *TA*, respectively. Then, the *saving* gained by *Merge*, denoted $\Delta_m(Q_i)$, is $\max\{T_e - T_m, 0\}$. The saving gained by *TA*, denoted $\Delta_{ta}(Q_i)$, is $\max\{T_e - T_{ta}, 0\}$.

Storing an index (RPL or ERPL) requires disk space. We denote by $S_{\text{RPL}}(Q_i)$ the required space for storing RPLs that support the evaluation of Q_i using *TA*. Note that only the part of the RPLs that is needed for computing the top-k elements must be stored, *i.e.,* the part that *TA* reads till reaching the stopping condition. We denote by $S_{\text{ERPL}}(Q_i)$ the disk space that is used when storing the ERPLs required for computing the answer to Q_i. We present now two approaches for choosing the lists to store, given that no more than d disk space should be used. Our goal is to maximize the time saving, for the different queries, weighted according to the frequencies in the workload.

4.1 Using Linear Programming

We describe now a mechanism for choosing indexes using boolean linear programming. Given a workload of queries $Q_1, \ldots, Q_l$ with frequencies $f_1, \ldots, f_l$, we generate the following set of linear equations.

Let x_{i1} be 1 when an ERPL is generated for Q_i and 0 otherwise. Similarly, let x_{i2} be 1 when the system stores an RPL for Q_i, and be 0 otherwise. Our goal is to maximize the weighted sum

$$\Sigma_{i=1}^{l}(x_{i1} f_i \Delta_m(Q_i) + x_{i2} f_i \Delta_{ta}(Q_i))$$

subject to

$$x_{i1} + x_{i2} \leq 1 \quad (1)$$

$$\Sigma_{i=1}^{l}(x_{i1} S_{\text{RPL}}(Q_i) + x_{i2} S_{\text{ERPL}}(Q_i)) \leq d \quad (2)$$

$$x_{ij} \in \{0, 1\}, \text{ for } 1 \leq i \leq l,\ j = 1, 2 \quad (3)$$

This linear-programming problem can be solved using known techniques such as the branch-and-cut or branch-and-bound algorithms. The actual time savings and disk space for typical queries should be measured experimentally and assigned in the formulas.

4.2 The Greedy Approach

Since boolean linear programming is known to be intractable, it should be used only when the number of queries in the workload is small. For other cases, we present a greedy algorithm that may not provide the optimal solution, but is guaranteed to provide a 2-approximation of the optimal solution.

In the greedy approach, we iteratively add indexes. Each time we add the index that seems to provide the largest improvement, *i.e.,* the highest ratio of the reduction in time to the addition of space. Suppose that I is the current set of stored indexes (RPLs and ERPLs). Consider a query Q_i that cannot be evaluated using merely I. Let I'_m be the minimal addition to ERPLs such that Q_i can be computed

using I and I'_m. Let I'_{ta} be the minimal addition to RPLs that enables the evaluation of Q_i over the indexes (*i.e.,* using I and I'_{ta}). Given that d is the available disk space, the *gain-cost ratio* of supporting Q_i for *Merge* is $f_i \frac{\Delta_m(Q_i)}{|I'_m|}$ if $|I'_m| \leq d$ and it is 0 otherwise (we denote by $|I'_m|$ the size of I'_m). Similarly, the gain-cost ratio for *TA* is $f_i \frac{\Delta_{ta}(Q_i)}{|I'_{ta}|}$ if $|I'_{ta}| \leq d$ and it is 0 otherwise.

In the greedy approach, in each iteration, the index that is added is the one that provides the highest, non zero, gain-cost ratio. Indexes are added until all the queries are supported or all the possible gain-cost ratios are zero.

Given a workload W, a set I_o of indexes is *optimal* if *(1)* $|I_o| \leq d$ (*i.e.,* I_o fits into the disk space d), and *(2)* for each I' with $|I'| \leq d$ holds

$$\Sigma_{i=1}^{l} f_i \cdot \text{Time}(Q_i, I_o) \leq \Sigma_{i=1}^{l} f_i \cdot \text{Time}(Q_i, I')$$

where $\text{Time}(Q_i, I)$ is the time for evaluating Q_i using the set of indexes I.

The following theorem shows that the greedy algorithm provides time saving that is, at the worst case, half the time saving provided by the optimal index creation. That is, the greedy algorithm is a 2-approximation algorithm.

Theorem 4.2 *Given a workload W, let I_o be the optimal set of indexes for W, generating time saving $T_o = \Sigma_{i=1}^{l} f_i \cdot (T_e(Q_i) - \text{Time}(Q_i, I_o))$ w.r.t. ERA. Let I_G be the set of indexes produced by the greedy algorithm, providing time saving $T_G = \Sigma_{i=1}^{l} f_i \cdot (T_e(Q_i) - \text{Time}(Q_i, I_G))$ w.r.t. ERA. Then, $T_o \leq 2 \cdot T_G$.*

5. Experimental Evaluation

We report below the performance of using different evaluation methods in *TReX* to show that none of the methods outperforms all the others.

5.1. Experimental Setup

We implemented *TReX* in Java and used BerkeleyDB (BDB) for the indexed tables. Our experiments were conducted over two collections of documents. One corpus is the IEEE collection provided in the INEX 2005 benchmark. This collection contains 16819 XML documents, and it has a size of 0.76 GB. For the IEEE collection, the sizes of the tables Elements and PostingLists, stored in BDB, are 1.52 GB and 8.05 GB, respectively. A second corpus is the Wikipedia collection provided in the INEX 2006 benchmark. The Wikipedia collection contains 659388 XML documents and has size of 4.6 GB. For Wikipedia, the sizes of the tables Elements and PostingLists, stored in BDB, are 3.91 GB and 48.1 GB, respectively.

We tested *TReX* on many INEX queries; however, we report here only the results of seven arbitrary queries that represent different behaviors of the evaluation methods. For measuring the running times of query evaluation, we conducted five separate runs starting with a cold Java Virtual Machine (JVM), for each query. The best and worst times were ignored and the reported runtime is the average of the remaining three times. The experiments were carried on a Windows XP Virtual Machine running on a 2.4GHz dual Opteron server, and the JVM was allocated 1 GB of RAM.

5.2. Experimental Results

The seven queries for which we present the evaluation performances of our methods are shown in Table 1. For each query, its ID is the ID used in INEX.

The graphs in Figures 4–6 show the evaluation times of queries, for the different methods. Each graph contains the evaluation time for computing all the answers to the query using *ERA* and *Merge*. In addition, the evaluation time for computing the top-k answers using *TA*, as a function of k is presented. The graphs also include a fourth method, denoted *ITA*, that refers to a *TA* with an ideal heap management. We present *ITA* because the management of the heaps in *TA* has a substantial influence on the running time. In *ITA*, we consider the operations of inserting an element to a heap or removing an element from a heap as being done in zero time (*i.e.,* we pause our time measure during these operations).

The evaluation times of the different methods for Query 202 appear in the left half of Figure 4 (the inset on the left shows times in logscale for small k values). For this query, computing using *Merge* an answer that contains all the 72269 elements that are relevant for this query is done in less than 10 seconds. Computing using *TA* the top-k values, for k between 50 to 30000, requires time of almost 1500 seconds. This time is close to the evaluation time needed for computing all the answers to the query using *ERA* (about 2000 seconds) and thus may not justify the cost of storing the additional redundant RPLs. Notice that having an ideal heap management could improve *TA* dramatically in this case. Also note that for large values of k the evaluation using *TA* is much more efficient than for small values of k. This is because for large k values, the heap holding the top-k values is large. Thus, most of the elements that are inserted into this heap are not being removed from it later on. This reduces the number of removal of element from the top-k heap, and thus, reduces the heap-management costs.

For Query 203, the evaluation times are depicted in the right half of Figure 4. Here *TA* is much more efficient than *ERA* (around 1000 seconds for computing all the 35624 answers using *ERA* versus around 100 seconds, at the worst case, when using *TA*. In this query, using an ideal heap

Query ID	NEXI Expression	Collection	# sids	# terms	# answers
202	//article[about(., ontologies)]//sec[about(., ontologies case study)]	IEEE	11	3	72269
203	//sec[about(., code signing verification)]	IEEE	11	3	35624
233	//article[about (.//bdy, synthesizers) and about (.//bdy, music)]	IEEE	2	2	1450
260	//bdy//*[about(., model checking state space explosion)]	IEEE	2	5	26750
270	//article//sec[about(., introduction information retrieval)]	IEEE	11	3	75309
290	//article[about(., "genetic algorithm")]	Wiki	6	2	3257
292	//article//figure[about(., Renaissance painting Italian Flemish -French -German)]	Wiki	1469	6	458

Table 1. NEXI queries we experimented with, the size of their translation and the size of the result.

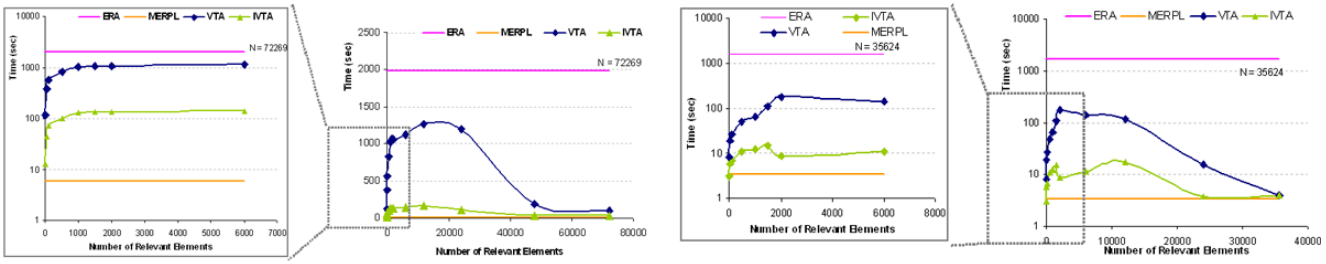

Figure 4. Evaluation times for Query 202 (left) and 203 (right).

could improve *TA* to be almost as good as *Merge* and for k values that are smaller than 10 even better than *Merge*. Thus, in this case if RPLS to support the query already exist, additional ERPLS may be redundant.

The evaluation times for Query 260 are shown on the left side of Figure 5. In some sense, this query shows a typical behavior of the methods. For very small k values (here, for $k \leq 10$), *TA* is the most efficient method. For larger values of k, *Merge* is much more efficient than *TA*—less than 10 seconds to compute all the answers using *Merge* in comparison to about 300 seconds when using *TA* for k up to 15000. As the size of k increases, the efficiency of *TA* gets closer to the efficiency of *ITA* since the cost of the heap management decreases. Except for small k values, the runtime of *ITA* is higher than the runtime of *Merge*. Also note that as k increases, the evaluation time of *ITA* increases. Thus, for large k values, *Merge* is better than *ITA*.

The graph on the right of Figure 5 shows that for Query 270 the size of k can drastically affect the running times of *TA*. While for large k values (above 70000) the evaluation time is around 20 seconds, it exceeds 800 seconds for certain k values. Hence, the added value of maintaining a redundant index for supporting a top-k query depends heavily on the size of k.

Figure 6 shows, from left to right, the graphs for Query 233, Query 290 and Query 292. For Query 233, *TA* and *Merge* are much more efficient than *ERA*. (They return the answer in less than a second while using *ERA* requires almost 1000 seconds.) Furthermore, *TA* is more efficient than *Merge* even when using a heap that is not ideal. Note that the translation of this query contains only two sid's and only two terms. Query 290 is an uncommon case where although *Merge* is usually more efficient than *TA*, for k values that are larger than 2500 the runtime of *TA* is smaller than the runtime of *Merge*. Query 292 has many sids but only a few answers. Not surprisingly, *ERA* is very inefficient in this case, *Merge* and *TA* are very efficient, where *TA* is slightly more efficient than *Merge*.

It has been proved that *TA* is optimal in the sense that its reads from the RPLS only the tuples that are necessary for computing the top-k elements [6]. That is, any deterministic algorithm will need to read from the RPLS at least as many tuples as *TA* does, when computing the top-k elements of the result. So, why is it that in many cases *Merge* is more efficient than *TA*? A key point for understanding why this happens is the fact that all the five queries over the IEEE collection (202, 203, 233, 260, 270) read the entire RPLS for $k \geq 10$. The same is true for the queries over the Wikipedia collection (290, 292), except that it happens for $k \geq 50$. When reading the entire lists, checking for the stopping condition of *TA* and managing the heap (for the top-k elements) reduces the efficiency of the query. Thus, *TA* is not as efficient as *Merge* in such cases.

6. Conclusion

We presented *TReX*, an XML-retrieval system that uses summaries, inverted lists and (sid, term, score) indexes for efficient evaluation of retrieval queries. We have shown how *TReX* evaluates NEXI queries using summaries and three evaluation methods (*ERA*, *TA* and *Merge*). We have also described an approach to self-managing indexes that enables choosing efficient methods given a workload of top-k

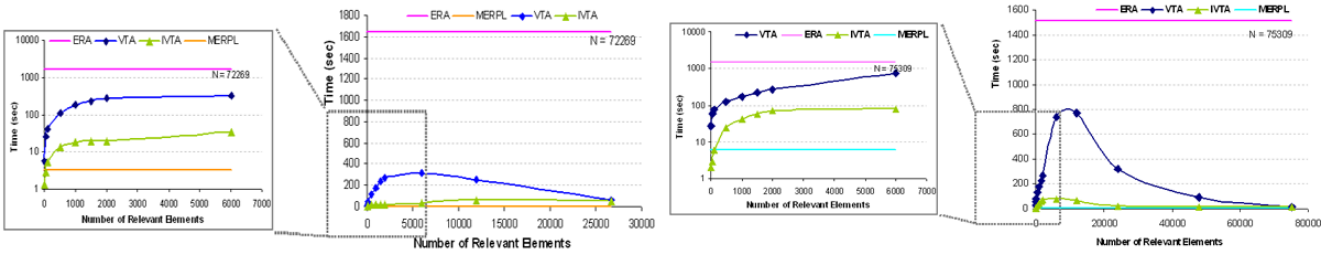

Figure 5. Evaluation times for Query 260 (left) and 270 (right).

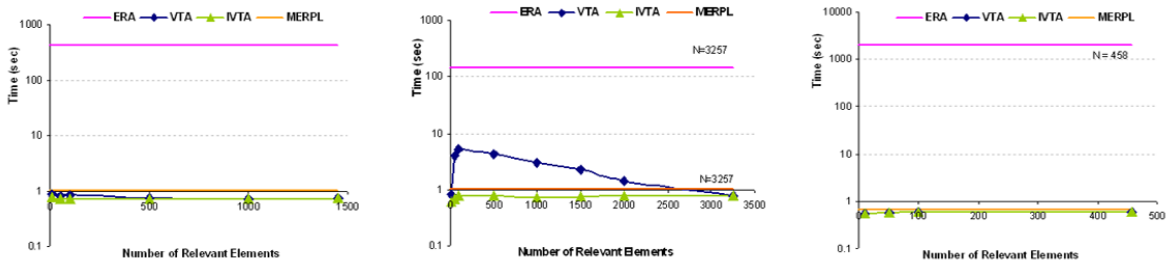

Figure 6. Evaluation times for Query 233 (left), 290 (center) and 292 (right).

NEXI queries, while minimizing disk space usage. Two algorithms are presented for choosing which indexes to store. One algorithm uses boolean linear programing for finding the optimal solution. An efficient 2-approximation greedy algorithm is also described. The experiments reported validate that, for computing top-k queries, relying on a single retrieval strategy is inferior to employing several strategies.

References

[1] XPath 1.0. `http://www.w3. org/TR/xpath`, 1999.

[2] S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying structured text in an XML database. In *SIGMOD*, 2003.

[3] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. Texquery: a full-text search extension to XQuery. In *WWW*, 2004.

[4] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. Flexpath: flexible structure and full-text querying for XML. In *SIGMOD*, 2004.

[5] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A semantic search engine for XML. In *VLDB*, 2003.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[7] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[8] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over xml documents. In *SIGMOD*, 2003.

[9] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on xml graphs. In *ICDE*, 2003.

[10] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[11] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD*, 2004.

[12] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, 2002.

[13] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in XML. In *ICDE*, 2005.

[14] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[15] F. Rizzolo and A. O. Mendelzon. Indexing XML data with ToXin. In *WebDB*, 2001.

[16] T. Schlieder and H. Meuss. Querying and ranking XML documents. *JASIST 2002*, 53(6):489–503, 2002.

[17] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for topx search. In *VLDB*, 2005.

[18] Andrew Trotman and Borkur Sigurbjornsson. Narrowed extended XPath I (NEXI). In *INEX*, 2004.