An Efficient and Versatile Query Engine for TopX Search

Martin Theobald, Ralf Schenkel, Gerhard Weikum

Max-Planck Institute for Informatics
Stuhlsatzenhausweg 85, D-66123 Saarbruecken, Germany
{mtb, schenkel, weikum}@mpi-inf.mpg.de

Abstract

This paper presents a novel engine, coined *TopX*, for efficient ranked retrieval of XML documents over semistructured but non-schematic data collections. The algorithm follows the paradigm of threshold algorithms for top-k query processing with a focus on inexpensive sequential accesses to index lists and only a few judiciously scheduled random accesses. The difficulties in applying the existing top-k algorithms to XML data lie in 1) the need to consider scores for XML elements while aggregating them at the document level, 2) the combination of vague content conditions with XML path conditions, 3) the need to relax query conditions if too few results satisfy all conditions, and 4) the selectivity estimation for both content and structure conditions and their impact on evaluation strategies. TopX addresses these issues by precomputing score and path information in an appropriately designed index structure, by largely avoiding or postponing the evaluation of expensive path conditions so as to preserve the sequential access pattern on index lists, and by selectively scheduling random accesses when they are cost-beneficial. In addition, TopX can compute approximate top-k results using probabilistic score estimators, thus speeding up queries with a small and controllable loss in retrieval precision.

1 Introduction

1.1 Motivation

Non-schematic XML data that comes from many different sources and inevitably exhibits heterogeneous structures and annotations (i.e., XML tags) cannot be adequately searched using database query languages like XPath or XQuery. Often, such queries either return too many or too few results. Rather the ranked-retrieval paradigm is called for, with relaxable search conditions and quantitative relevance scoring. Note that the need for ranking goes beyond adding Boolean text-search predicates to XQuery.

Research on applying IR techniques to XML data has started five years ago [13, 17, 35, 36] and has meanwhile gained considerable attention. The emphasis of the current paper is on efficiently supporting vague search on element names and terms in element contents in combination with XPath-style path conditions. A typical example query could be phrased in the NEXI language used for the INEX benchmark [24] as follows:

```
//book[about(.// "Information Retrieval XML")]
       //[about(.//affiliation "Stanford") and
          about(.//reference "Page rank")].
```

This twig query should find the *best matches* for books that contain the terms "Information Retrieval XML" and have descendants tagged as affiliation and reference with content terms "Stanford" and "Page rank", respectively. The challenge lies in processing such queries *efficiently*. The method of choice for top-k similarity queries is the family of threshold algorithms, developed by [16, 20, 32] and related to various heuristics for processing index lists in IR [9, 33, 6]. These methods scan index lists for terms or attribute values in descending order of local (i.e., per term) scores and aggregate the scores for the same data item into a global score, using a monotonic score aggregation function such as (weighted) summation. Based on clever bookkeeping of score intervals and thresholds for top-k candidate items, index scans can often terminate early, when the top-k items are determined, and thus, the algorithm often only has to scan short prefixes of the inverted lists. In contrast to the heuristics adopted by Web search engines [29], the threshold algorithms compute exact results and are provably optimal in terms of asymptotic costs. The XML-specific difficulties arise from the following issues:

- Scores and index lists refer to individual XML elements and their content terms, but we want to aggregate scores at the document level and return

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

documents or XML subtrees as results, thus facing two different granularities in the top-k query processing.
- Good IR scoring models for text documents cannot be directly carried over, because they would not consider the specificity of content terms in combination with element or attribute tags. For example, the term "transactions" is viewed as specific when occurring within elements of type `<section>` or `<caption>` but is non-informative in a `<journalname>`.
- Relevant intermediate results of search conditions must be tested as to whether they satisfy the path conditions of the query, and this may incur expensive random accesses to disk-resident index structures.
- Instead of enforcing a conjunctive query processing, it is desirable to relax path conditions and rather rank documents by a combination of content scores and the number of structural query conditions that are satisfied.
- An efficient query evaluation strategy and the pruning of result candidates must take into consideration the estimation of both aggregated scores and selectivities of path conditions.

A viable solution must reconcile local scorings for content search conditions, score aggregation, and path conditions. As a key factor for efficient performance, it must be careful about random accesses to index structures, because random accesses are one or two orders of magnitude more expensive than (the amortized cost of) a sequential access. It should exploit precomputations as much as possible and may utilize the technology trend of fast growing disk space capacity (whereas disk latency and transfer rates are improving only slowly). The latter makes redundant data structures attractive, if they can be selectively accessed at query run-time.

1.2 Related Work

Efficient evaluation of XML path conditions is a very fruitful research area. Solutions include structural joins [3], the multi-predicate merge join [41], the staircase join based on index structures with pre- and postorder encoding of elements within document trees [19] and holistic twig joins [8, 25]; the latter is probably the most efficient method for twig queries using sequential scans of index lists and linked stacks in memory. [27] extends XQuery to support partial knowledge of the schema. None of these papers considers result ranking and finding the top-k results only.

Information retrieval on XML data has become popular in recent years. Some approaches extend traditional keyword-style querying to XML data [14, 21, 22]. [17, 13, 36] introduced full-fledged XML query languages with rich IR models for ranked retrieval. [11] and [18] developed extensions of the vector space model for keyword search on XML documents. [35] addressed vague structural conditions, and [5] combined this theme with full-text conditions. [4] introduced a query algebra for XML queries that integrates IR-style query processing. To the best of our knowledge, among these systems XRANK [21] is the only one that provides efficient support for finding the top-k results, but only for keyword queries.

Top-k queries have been tackled in various application settings such as multimedia similarity search, spatial data analysis, information retrieval in digital libraries and the Web, or preference queries on product catalogs and other kinds of structured databases [2, 10, 12, 15, 16, 20, 23, 32, 38]. The TA (threshold algorithm) family [16, 20, 32] has evolved as the most efficient and versatile method; we discuss it in more detail in Section 2. Recent work on making XML ranked retrieval more efficient has been carried out by [26] and [31]. [26] uses path index operations as basic steps; these are invoked within a TA-style top-k algorithm. The scoring model can incorporate distance-based scores, but the experiments in the paper are limited to DB-style queries rather than XML IR in the style of the INEX benchmark. [31] focuses on efficient evaluation of approximate structural matches along the lines of [5]. The paper considers primarily structural similarity by means of outer joins, and disregards optimizations for content term search. The prior work that is closest to the TopX engine is [26]; our performance studies in Section 7 compare TopX against this work. Our own prior work on XXL [36, 37] did not use a top-k algorithm; TopX has a radically different query processing architecture and outperforms XXL by a large margin.

1.3 Contributions

This paper presents the TopX algorithm and its implementation in a prototype search engine for ranked retrieval of XML, supporting the NEXI language of the INEX benchmark [24] and extending search along all XPath axes with IR-style search conditions. The salient properties of TopX and the paper's novel contributions are the following:

- It efficiently processes XML IR queries with support for all XPath axes, carefully designed index structures, efficient priority queue management for top-k candidates, and judicious scheduling of expensive random accesses.
- It supports probabilistic pruning of candidates, along the lines of [38] but considerably extending these prior techniques to an XML setting, to gain additional speed at the expense of a small loss in precision of the top-k results.
- It uses novel techniques for estimating aggregated scores of candidates as well as selectivities of both tag-term content conditions and structural path conditions which are used to drive the scheduling decisions on random accesses to index lists.
- Our XML-specific scoring model takes into consideration both statistics about tag-term combinations in the underlying XML corpus and the compactness of subtrees that satisfy the search conditions.
- The paper provides an extensive performance evaluation on various real-life, fairly large datasets: the INEX benchmark, the IMDB structured movie

database, and the TREC 2004 Terabyte track.

2 Computational Model

We consider a simplified XML data model, where idref/XLink/XPointer links are disregarded. Thus every document forms a tree of nodes, each with a *tag* and a *content*. We treat attributes as children of the corresponding node. With each node, we can additionally associate its *full content* which is defined as the concatenation of the contents of all the node's descendants.

Our query processing methods are based on *precomputed index lists* that are sorted in descending order of appropriately defined scores for individual tag-term content conditions, and our algorithmic rationale for top-k queries follows that of the family of *threshold algorithms (TA)* [16, 20, 32]. For finding the matches to multiple search conditions (e.g., multiple tag-term conditions), scoring, and ranking them, TA scans all relevant index lists in an interleaved manner. In each scan step, when the algorithm sees the score for a data item in one list, it combines this score with scores for the same data item previously seen in other index lists into a *global score* using a monotonic aggregation function such as weighted summation.

The TA family of algorithms comes in two flavors: a) with random accesses to data items in addition to the sorted scans on index lists, and b) with sorted access only. Hybrid methods that aim to schedule random accesses at appropriate points during query execution have been proposed as well; TopX proposes an XML-specific cost model to schedule random lookups to both content-related and structural query conditions.

Sorted access benefits from sequential disk I/O with asynchronous prefetching and high locality in the processor's cache hierarchy; so it has much lower amortized costs than random access. Threshold algorithms with eager random accesses look up the scores for a data item in all query-relevant index lists, when they first see the data item in one list. Thus, they can immediately compute the global score of the item, and need to keep in memory only the current top-k items with their scores. Algorithms with a focus on sorted access do not eagerly look up all candidates' global scores and therefore need to maintain a candidate pool in memory, where each candidate is a data item d that has been seen in at least one list and may qualify for the final top-k result based on the following information (we denote the score of data item d in the i-th index list by $s_i(d)$, and we assume for simplicity that the score aggregation is summation):

- the set $E(d)$ of evaluated lists where d has already been seen,
- the $worstscore(d) := \sum_{i \in E(d)} s_i(d)$ based on the known scores $s_i(d)$, and
- the $bestscore(d) := worstscore(d) + \sum_{i \notin E(d)} high_i$ that d could possibly still achieve based on $worstscore(d)$ and the upper bounds $high_i$ for the scores in the yet unvisited parts of the index lists.

The algorithm terminates when the *worstscore* of the rank-k in the current top-k result, coined *min-k*, is at least as high as the highest *bestscore* among all other candidates.

Algorithm 1 Baseline top-k algorithm

```
1: min-k := 0
2: for all index lists L_i (i = 1..m) do
3:    d := (did, s_i) //scan next d
4:    E(d) := E(d) ∪ {i};
5:    high_i := s_i;
6:    worstscore(d) := Σ_{i∈E(d)} s_i;
7:    bestscore(d) := worstscore(d) + Σ_{i∉E(d)} high_i;
8:    if worstscore(d) > min-k then
9:       replace min{worstscore(d')|d' ∈ top-k} by d;
10:      remove d from candidates;
11:   else if bestscore(d) > min-k then
12:      candidates := candidates ∪ d;
13:   else
14:      drop d from candidates if present;
15:   end if
16:   min-k := min{worstscore(d')|d' ∈ top-k};
17:   if (scanDepth % prefetchSize) = 0 then
18:      for all d' ∈ candidates do
19:         consider random accesses to L_i' for i' ∉ E(d')
               according to cost model;
20:         update bestscore(d') using current high_i;
21:         if bestscore(d') ≤ min-k
               or P[bestscore(d') > min-k] < ε then
22:            drop d' from candidates;
23:         end if
24:      end for
25:   end if
26:   if candidates = ∅
         or max{bestscore(d')|d' ∈ candidates} ≤ min-k then
27:      return top-k;
28:   end if
29: end for
```

The literature contains various extensions to this baseline algorithm. Among these, [38] discuss how to efficiently implement the candidate pool using priority queues with judicious queue maintenance. The same work also proposes to replace the conservative threshold test by a probabilistic test for an approximative top-k algorithm with probabilistic precision and recall guarantees. The most flexible implementation uses histograms to capture the score distributions in the individual index lists and computes convolutions of histograms in order to predict the probability that an item d has a global score above the min-k threshold:

$$P[\sum_{i \in E(d)} s_i(d) + \sum_{j \notin E(d)} S_j(d) > min\text{-}k \mid S_j(d) \leq high_j] < \epsilon,$$

where $S_j(d)$ denotes the random variable for the score of d in list j. When this probability for some candidate document d drops below a threshold ϵ (e.g., set to 0.1), d is discarded from the candidate set. The special case of $\epsilon = 0$ corresponds to the conservative threshold algorithm. Figure 1 shows pseudo code for this algorithm. The current paper builds on this method.

The pseudo code has an option for making random accesses when assessing a candidate and reducing the uncertainty in the $[worstscore, bestscore]$ interval, if it is desired and cost-beneficial at this point. This idea gives rise to a family of hybrid algorithms that allow random accesses but aim to minimize them or postpone them to the latest possible point [7, 12, 30].

All candidates that are of potential relevance for the final top-k result are collected in a hash table (the *cache*) in main memory; this data structure has the full

information about elements, *worstscores*, *bestscores*, etc. In addition, two priority queues merely containing pointers to these cache entries are maintained in memory and periodically updated. The top-k queue uses *worstscores* as priorities to organize the current top-k documents, and the candidate queue uses *bestscores* as priorities to maintain the condition for threshold termination. In this paper, we employ the following strategy for queue management, coined the "smart" strategy in [38]. We periodically rebuild the candidate queue and bound its length (e.g., to a few hundred entries), taking into account the most recent $high_i$ values from the index scans, and stop the index scans when the top-priority entry can no longer qualify for the top-k result (with high probability, if the score predictor is used).

3 System Architecture

3.1 Query Language

The TopX engine supports a variant of XPath that has been extended by IR-style conditions, namely the NEXI dialect [40] used in the INEX benchmark [24] (see Section 1). TopX currently supports the full NEXI specification. Like in XPath, a query is expressed by a location path that consists of at least one location step. In each location step, the set of qualifying nodes can be restricted with one or more predicates that can be Boolean combinations of keyword conditions (denoted by the so called *about* operator), e.g., `//movie[about(.//title, "Matrix 3")]//actor[about(.//, "Reeves")]`.

3.2 Content Scores

For content scoring, we make use of statistical measures that view the content or full content of a node n with tag A as a bag of words: 1) the *term frequency*, $tf(t,n)$, of term t in node n, which is the number of occurrences of t in the content of n; 2) the *full term frequency*, $ftf(t,n)$, of term t in node n, which is the number of occurrences of t in the full content of n; 3) the *tag frequency*, N_A, of tag A, which is the number of nodes with tag A in the entire corpus; 4) the *element frequency*, $ef_A(t)$, of term t with regard to tag A, which is the number of nodes with tag A that contain t in their full contents in the entire corpus. Now consider a content condition of the form `A//"`$t_1 \ldots t_m$`"`, where `A` is a tag name and t_1 through t_m are terms that should occur in the full contents of a subtree. The score of node n with tag `A` for such a condition should reflect

- a monotonic aggregation of the ftf values of the terms t_1 through t_m (or tf values, if we use the child rather than the self-or-descendant axis), thus reflecting the *relevance* of the terms for the node's content,
- the *specificity* of the search terms, with regard to $ef_A(t_i)$ and N_A statistics for all node tags, and
- the *compactness* of the subtree rooted at n that contains the search terms in its full content.

In the following we focus on the self-or-descendant axis (i.e., the full-content case) as the much more important case for XML IR with vague search; the case for the child axis follows analogously. Our scoring of node n with regard to condition `A//"`t_1 `...`t_m`"` uses formulas of the following type:

$$score(n, A//"t_1 \ldots t_m") := \frac{\sum_{i=1}^{m} relevance_i \cdot specificity_i}{compactness(n)},$$

where $relevance_i$ reflects ftf values, $specificity_i$ is derived from N_A and $ef_A(t_i)$ values, and $compactness(n)$ considers the subtree or element size for length normalization. Note that specificity is made XML-specific by considering combined tag-term frequency statistics rather than global term statistics only. It serves to assign different weights to the individual tag-term pairs, a common technique from probabilistic IR.

An important lesson from text IR is that the influence of the term frequency and element frequency values should be sublinearly dampened to avoid a bias for short elements with high term frequency of a few rare terms. To address these considerations, we have adopted the popular and empirically usually much superior Okapi BM25 scoring model (originating in probabilistic IR for text documents [34]) to our XML setting, leading to the following scoring function:

$$score(n, A//"t_1 \ldots t_m") := \sum_{i=1}^{m} \frac{(k_1+1) \cdot ftf(t_i,n)}{K + ftf(t_i,n)} \cdot \log\left(\frac{N_A - ef_A(t_i) + 0.5}{ef_A(t_i) + 0.5}\right)$$

$$\text{with } K = k_1 \left((1-b) + b \frac{\sum_{t \in full\ content\ of\ n} ftf(t,n)}{avg_{n'}\{\sum_{t'} ftf(t',n') \mid n' \text{ with tag } A\}}\right).$$

For more background on the theoretical underpinnings of Okapi BM25 see [34]. The modified Okapi function was applied to both XML collections in the experiments of this paper with the parameters k_1 and b set to 1.2 and 0.75, respectively, for all element types. With regard to overall retrieval quality, the above formula would also allow a more elaborated parameter optimization for individual element types, which would go beyond the subject of this work. Good scoring functions for XML, e.g., based on statistical language models, are an active research issue [24]; our Okapi-based scoring model achieved very good result quality on the INEX benchmark.

3.3 Structure Conditions

For efficient testing of structural conditions, we transitively expand all structural dependencies. For example, in the query `//A//B//C[.// "t"]` an element with tag `C` (and content term `"t"`) has to be a descendant of both `A` and `B` elements. This way, the query forms a *directed acyclic graph* with tag-term conditions or elementary tag conditions as nodes and all transitively expanded descendant relations as edges. Branching path expressions are expressed analogously. This transitive expansion of structural constraints is a key for efficient path validation and allows an *incremental testing* of path satisfiability. If `C` in the above example is not a valid descendant of `A`, we may safely prune the candidate document from the priority queue, if its *bestscore* falls below the current min-k threshold without ever looking up the `B` condition.

In non-conjunctive (aka. "andish") retrieval, a result document (or subtree) should still satisfy most structural constraints, but we may tolerate that some tag names or path conditions are not matched. This is useful when queries are posed without much information about the possible and typical tags and paths, e.g., when the XML corpus is a federation of datasets with highly diverse schemas. Our scoring model essentially counts the number of structural conditions (or tags) o_j that are still to be satisfied by a result candidate d and assigns a small and constant score mass c for every condition that is matched. This structural score mass is combined with the content scores and aggregated with each candidate's $[worstscore(d), bestscore(d)]$ interval. In our setup we have set $c = 1$, whereas content scores were normalized to $[0, 1]$, i.e., we emphasize the structural query conditions.

3.4 Database Schema and Indexing

Index lists are implemented as database tables; Figure 1 lists the corresponding schema definitions. Nodes in XML documents are identified by the combination of document id (`did`) and pre-order (`pre`). Navigation along all XPath axes is supported by both the `pre` and `post` attributes using the indexing technique by [19]. The actual index lists are processed by the top-k algorithm using the various B+-tree indexes that are created on the base tables.

```
 // Full content index for tag-term pairs
CREATE TABLE TagTermFeatures (
  did NUMBER, // Document id
  tag VARCHAR2(32), // Element name
  term VARCHAR2(32), // Term in element
  pre NUMBER, // Pre-order of element
  post NUMBER, // Post-order of element
  score NUMBER, // Full content score of tag-term pair
  maxscore NUMBER); // Max(score) of tag-term pair per doc
// Sorted access by (tag, term) with desc.maxscore
CREATE INDEX TagTermScoreIndexSorted ON
  Features(tag,term,maxscore,did,score,pre,post);
// Random access by (did, tag, term)
CREATE INDEX TagTermScoreIndexRandom ON
  Features(did,tag,term,score,pre,post);
// Navigational elements index
CREATE TABLE Elements (
  did NUMBER, // Document id
  tag VARCHAR2(32), // Element name
  pre NUMBER, // Pre-order of element
  post NUMBER ); // Post-order of element
// Random access on elements by (did, tag)
CREATE INDEX ElementIndexRandom ON
  Elements(did,tag,pre,post);
```

Figure 1: Tables for TopX index structures

The `TagTermFeatures` table contains the actual node contents indexed as one row per tag-term pair per document, together with their local scores (referring either to the simple content or the full content scores) (see Section 3.2) and their pre- and post-order numbers. For each tag-term pair, we also provide the *maximum score* among all the rows grouped by tag, term, and document id to extend the previous notion of single-line sorted accesses to a notion of *sorted block-scans*. TopX scans each list corresponding to the key $(tag, term)$ in descending order of $(maxscore, did, score)$, using the `TagTermScoreIndexSorted` index. Sequential scans prefetch all tag-term pairs for the same `did` in one shot and keep them in memory for further processing which we refer to as sorted block-scans, the reason will be discussed in Section 4. Random accesses to content scores for a given document, tag, and term are performed through range scans on the `TagTermScoreIndexRandom` index using the triple $(did, tag, term)$ as key. Structural tests to the `Elements` are performed through random accesses to the `ElementIndexRandom` index, only, using the key (did, tag).

We fully precompute and materialize the `TagTermFeatures` table to efficiently support the self-or-descendant axis. We propagate, for every term t that occurs in a node n with tag A, its local tf value "upwards" to all ancestors of n and compute the ftf values of these nodes for t. Obviously, this may create a redundancy factor that can be as high as the length of the path from n to the root.

4 TopX Query Processing

The query processor decomposes the query into content conditions, each of which refers to a tag-term pair, and into additional elementary tag conditions (e.g., for navigation of branching path queries), plus the path conditions that constrain the way how the matches for the tag-term pairs and elementary tag conditions must be connected. We concentrate on content conditions that refer to the self-or-descendant axis, i.e., the full contents of elements. This way, each term is connected to its last preceding tag in the location path, in order to merge each tag-term pair into a single query condition with a corresponding list in the inverted index. For each such content condition (e.g., `A[.//"a"]`), a sorted index scan on the `TagTermScoreIndexSorted` index is started. Note that by using tag-term pairs for the inverted index lookups, we immediately benefit from more selective, combined tag-term features and shorter index lists. The hypothetical combinatorial bound of $\#tags \cdot \#terms$ rows is by far not reached for any of the collections. There are three key issues to be addressed for efficiency:

1. The random accesses for tag-only conditions are expensive and should be minimized.
2. Path conditions that test partial results for their connectivity along the specified XPath axes must be inexpensive to evaluate.
3. We would like to achieve fast convergence of *worstscores* and *bestscores* of candidate items to prune candidates and terminate index scans as early as possible.

As for the first issue, TopX postpones random accesses as much as possible, namely, until the point when random I/Os are cost-beneficial according to our scheduling approach (see Section 6) for a given candidate. For most candidates the content-based *bestscore* quantiles

(or the probabilistically estimated quantiles) already become low enough, so they can be dropped from the candidate queue before the structural conditions need to be evaluated. This results in major savings of random accesses.

The second requirement is met by adding pre- and post-order values to the entries in the main index lists (the `TagTermFeatures` table, see Section 3.4), following the XPath accelerator work by [19]. This gives us an efficient in-memory test whether an element e_1 is an ancestor of another element e_2 (within the same document) by evaluating $pre(e_1) < pre(e_2) \wedge post(e_1) > post(e_2)$. There are analogous tests for all XPath axes, including the child and preceding axes by extending the schema with the *level* information. Since the pre/post information is at hand as we fetch entries from the index lists, we can test path conditions for candidates with high local scores without additional index accesses. If an element fails a path test, we simply drop it and exclude it from the structural join.

The third requirement, fast convergence of *worstscores* and *bestscores* for entire documents, can be met by eagerly eliminating uncertainty about matches among the elements of a document for which only a subset of elements has been seen so far in the index scans. The problem is that we may have to keep the document in the candidate queue for a long time until we finally find low-score elements that satisfy the content conditions but violate the path conditions. This would keep up the *bestscores* unnecessarily high and render the candidate pruning ineffective. To efficiently fetch all tag-term pairs connected to a content condition in the query, we perform *sorted block-scans* that fetch *all* the elements being relevant for a content condition from the respective index list in an inexpensive series of sequential disk I/O *per document*. The `TagTermScoreIndexSorted` that materializes this block grouping of elements in descending order of $(maxscore, did, score)$ for each tag-term condition helps us to this end. Analogously to the single-line sorted accesses model, the *maxscore* attribute now becomes the basis for the $high_i$ values used in each index list to estimate the upper *bestscore* bounds for all candidates. Note that the $high_i$ values now serve as a more generous upper bound for the score each candidate can still achieve, but our experiments indicate that pruning remains effective.

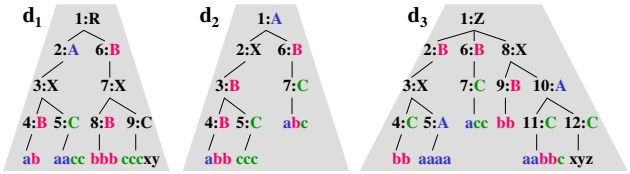

Figure 2: XML example documents

4.1 Query Processing Example

As an illustration of the query processing, consider the example data in Figure 2 and the following twig query: `//A[.//B[.//"b"] and .//C[.//"c"]]`. Figure 3 shows an excerpt of the respective `TagTermScoreIndexSorted` index. For simplicity, we do not show the Okapi-based scores here, but rather pretend that our scores are mere ftf values normalized by the number of terms in a subtree (e.g., for the tag-term pair `A:a`, the element e_{10} in document d_3 has score 1/4, because the term a occurs twice among the eight terms under e_{10}). TopX evaluates the query by opening index scans for the two tag-term pairs `B:b`, and `C:c`, and block-fetches the best document for each of the two conditions. For example, for `B:b`, the first three lines of index list L_2 in Figure 3 that belong to the same document d_1 are fetched as a sorted block-scan.

Figure 2 gives pseudo code for the TopX algorithm. As the index scans proceed in a baseline round-robin fashion among all index lists L_i connected to a content condition (i.e., tag-term pairs), the algorithm continuously computes $[worstscore(d), bestscore(d)]$ intervals for each candidate d that it is fetched by a sorted access and periodically updates the $bestscore(d)$ values of all candidates using the current $high_i$ values and the score mass of the not yet evaluated structural constraints $o_j \cdot c$. As the first round of block-scan fetches yields two different documents, the algorithm needs to continue fetching the second-best document for each condition. After the second round, it happens that d_3's relevant elements for both conditions are in memory at this point. A random access for all `A` elements in d_3 can now be triggered, if it is cost-beneficial (see Section 6). If so, we can efficiently test both path conditions for d_3, namely whether a `B:b` element is a descendant of an `A` element and a `C:c` element is a descendant of the same `A` element, by comparing the pre- and post-order numbers of the respective element pairs. This way, it is detected that none of d_3's element triples satisfies both path conditions and the $wortscore(d_3)$ and $bestscore(d_3)$ values have both converged to the final value $1 + 2/3$, however, without the score mass $c = 1$ for the missed `A` condition. The same test can be performed for document d_1 at this point, but only for one of the two path conditions, namely, whether an `A` element has a `B:b` element among its descendants. The second condition, namely the connecting between `A` and `C:c`, can be tested only later, when the matches for `C:c` within d_1 are encountered on the `C:c` index list. As d_1 has valid element pairs after the `A` vs. `B:b` test, we recompute the $[worstscore(d_1), bestscore(d_1)]$ interval which now becomes $[1+1, 1+1+3/5]$. If the $worstscore(d_1) > min\text{-}k$, we put d_1 into the top-k results; if otherwise the $bestscore(d_1) > min\text{-}k$, we put d_1 into the candidate queue and optionally perform the probabilistic threshold test as to whether d_1 still has a good chance to qualify for the top-k.

5 Score and Selectivity Predictors

For score prediction and probabilistic candidate pruning along the lines of [38], histograms on score distributions are stored in a separate database table and the histograms that are relevant for each content-

L_i	tag	term	maxscore	did	score	pre
1	A	a	1	d3	1	e5
	A	a	1	d3	1/4	e10
	A	a	1/2	d1	1/2	e2
	A	a	2/9	d2	2/9	e1
2	B	b	1	d1	1	e8
	B	b	1	d1	1/2	e4
	B	b	1	d1	3/7	e6
	B	b	1	d3	1	e9
	B	b	1	d3	1/3	e2
	B	b	2/3	d2	2/3	e4
	B	b	2/3	d2	1/3	e3
	B	b	2/3	d2	1/3	e6
3	C	c	1	d2	1	e5
	C	c	1	d2	1/3	e7
	C	c	2/3	d3	2/3	e7
	C	c	2/3	d3	1/5	e11
	C	c	3/5	d1	3/5	e9
	C	c	3/5	d1	1/2	e5

Figure 3: Block index for the example of Figure 2

Algorithm 2 Structure-Aware Top-k Algorithm

```
1: min-k := 0
2: for all index lists L_i (i = 1...m) do
3:    e := (maxscore, did, score, pre, post); //block-scan e in d
4:    high_i := e.maxscore;
5:    elements_i(d) := elements_i(d) ∪ e;
6:    for all i' ∈ E(d) with i' ≠ i do
7:       for all e' ∈ elements_i'(d) do
8:          incrementally test path constraints on (e, e')
            using e.pre, e.post, e'.pre, and e'.post;
9:       end for
10:      if e does not match any connected e' then
11:         drop e from elements_i(d);
12:      end if
13:   end for
14:   E(d) := E(d) ∪ {i};
15:   worstscore(d) := Σ_{i∈E(d)} max{e.score|e ∈ elements_i(d)};
16:   bestscore(d) := worstscore(d) + Σ_{i∉E(d)} high_i + o_j · c;
17:   min-k := min{worstscore(d')|d' ∈ top-k};
18:   //continue as in baseline top-k algorithm shown in Fig.1
19: end for
```

related query dimension are loaded on-demand in a single database prefetch prior to query execution. We chose single-dimensional equi-width histograms over tag-term pairs for simplicity, with a default of 100 cells per histogram. Predicting the score summation over multiple index lists requires a convolution of histograms; this is performed on demand during query run-time. We use one histogram for each tag-term pair according to our *full content* scoring model (see Section 3.2). This way we capture the joint tag-term distribution of scores which is important as some terms have high scores in combination with particular tags (e.g., the term "transactions" in elements named "journal-name" vs. elements named "section").

The convolution that is needed for score predictions is dynamically performed during query execution whenever the priority queue of candidates is rebuilt. Note that we merely need to precompute histograms for frequent tag-term pairs in the collection, since the first pruning step is applied after B sorted accesses. This approach yields about 75,000 histograms for INEX (and 118,000 for IMDB) for tag-term pairs (A, a_i) with $ef_A(a_i) > 100$. There is no necessity to maintain histograms for non-frequent combinations, because they will be scanned through after the first batch of scan steps anyway.

To consider if and when we should perform random accesses to look up tag-term scores for particular candidates, TopX also needs selectivity estimators and combines it with the score predictor. We introduce Bernoulli random variables X_i for the i^{th} tag-term condition being satisfied by candidate d, and and we estimate the combined probability that d satisfies the conditions for which d has not yet been seen during the index scans *and* will eventually have a sufficiently high total score to qualify for the top-k result as $q(d) :=$

$$P[\sum_{j=1}^{m} S_j > \textit{min-k} | S_j \leq high_j] \cdot P[X_i = 1 \textit{ for all } i \notin E(d)]$$

(assuming independence between the S_j and the X_i variables, for tractability). A perfectly analogous approach can be used to estimate, if a candidate will have a high total score and satisfies certain structural conditions; we will come back to this aspect in the next section.

We can estimate $P[X_i]$ for each i separately by the ratio $P[X_i] = \frac{l_i}{n}$, where $l_i := n_i - p_i$ is the length n_i of index list L_i minus the current index scan position p_i in this list and n is the total number of documents in the corpus. We could then assume independence among the X_i variables, but this would lead to a very crude selectivity estimator. Instead we capture the covariance cov_{ij} of the (t_i, t_j) pairs for all i, j that appear together in prior query histories or have strong correlation in the data collection. We can then use the equality $P[X_i \wedge X_j] = P[X_i]P[X_j] + cov_{ij}$ and the chain rule to derive:

$$P[X_i = 1 \textit{ for all } i \in M] = \prod_{i \in M} \frac{P[X_i]P[X_{i+1}] + cov_{i,i+1}}{P[X_i]},$$

where M is a set of conditions to be satisfied.

6 Random Access Scheduling

The rationale of TopX is to postpone expensive random accesses as much as possible and perform them only for the best top-k candidates. However, it can be beneficial to test path conditions earlier, namely, for eliminating candidates that do not satisfy the conditions but have high *worstscores*. Moreover, in the query model where a violated path condition leads to a score penalty, positively testing a path condition increases the *worstscore* of a candidate, thus potentially improving the min-k threshold and leading to increased pruning subsequently. In TopX, we consider random accesses at specific points only, namely, whenever the priority queue is rebuilt. At this point, we consider each candidate d and decide whether we should make random accesses to test unresolved path conditions, or look up missing scores for content conditions. For this scheduling decision, we have developed two different strategies.

6.1 MinProbe Scheduling

The first strategy, coined *MinProbe*, aims at a minimum number of random accesses by probing structural conditions for the most promising candidates, only. Since we do not perform any sorted scans on elementary tag conditions, we treat structural conditions as *expensive predicates* in the sense of [12]. We

schedule random accesses only for those candidates d whose $worstscore(d) + o_j \cdot c > min\text{-}k$, where o_j is the number of untested structural conditions for d and c is a static score mass that d earns with every satisfied structural condition (see Section 3.3).

6.2 BenProbe Scheduling

The second strategy, coined *BenProbe* uses an analytic cost model. We denote the number of documents in the priority queue by Q, and the batch size for the next round of sorted accesses on the index lists by B. The probability that document d, which has been seen in the tag-term index lists $E(d)$ and has not yet been encountered in lists $\overline{E}(d) := [1..m] - E(d)$, qualifies for the final top-k result is estimated by the combined score predictor (see Section 5) and denoted as $q(d)$. We estimate the selectivity of the o_j remaining path or twig conditions by precomputed frequencies of ancestor-descendant and twig patterns, i.e., pairs and triples of tag names. This is a simple form of XML synopsis; it could be replaced by more advanced approaches like [1, 28].

BenProbe compares the cost of making random accesses to tag-term index lists or to indexes for structural path conditions versus the cost of proceeding with the sorted-access index scans. For all three cost categories, we consider only the *expected wasted cost (EWC)* which is the expected number of random (or sorted) accesses that our decision would incur but would not be made by an optimal schedule. For looking up unknown scores of a candidate d in the index lists $\overline{E}(d)$, we would incur $|\overline{E}(d)|$ random accesses which are wasted, if d does not qualify for the final top-k result (even after considering the additional score mass from $E(d)$). By computing the convolution histogram for $\overline{E}(d)$, we can estimate this probability, using the current min-k threshold, as $P[d \notin final\ top\text{-}k] = 1 - P[S(d) > min\text{-}k] = 1 - P[\sum_{i \in \overline{E}(d)} S_i > min\text{-}k - worstscore(d)] =: 1 - q(d)$. Then the random accesses to resolve the missing tag-term scores have expected wasted cost:

$$EWC\text{-}RA1(d) \quad = \quad (1 - q(d)) \cdot |\overline{E}(d)| \quad [RA].$$

As for path conditions, the random accesses to resolve all o_j path conditions are "wasted cost" if the candidate does not make it into the final top-k which happens, if the number of satisfied conditions is not large enough to accumulate enough score mass. The probability that d qualifies for the final top-k is estimated as $\sum_{Y' \subseteq Y} P[o'\ conditions\ i_1 \ldots i_{o'}\ are\ satisfied] \cdot P[\sum_{i \in \overline{E}(d)} S_i > min\text{-}k - worstscore(d) - o' \cdot c] =: q'(d)$, where the sum ranges over all subsets Y' of Y for the o_j conditions. $P[Y\ is\ satisfied]$ is estimated as $\prod_{\nu \in Y} p_\nu \cdot \prod_{\nu \notin Y} (1 - p_\nu)$, assuming independence for tractability. The independence assumption can be relaxed by the covariance-based technique mentioned in Section 5. For efficiency, rather than summing up over the full amount of subsets $Y' \subseteq Y$, a lower-bound approximation can be used. Then the random accesses for path and twig conditions have expected wasted cost:

$$EWC\text{-}RA2(d) \quad = \quad \big(1 - q'(d)\big) \cdot o_j \quad [RA].$$

For a candidate d, the sorted accesses are wasted, if we do not learn any new information about the total score of d, that is, when we do not encounter d in any of the m lists. The probability of *not* seeing d in the i^{th} list in the next B steps is $1 - \frac{l_i}{n}\frac{B}{l_i} = 1 - \frac{B}{n}$. Assuming independence, the probability of not seeing d at all in the next round of index scan steps then becomes $\prod_{i \in \overline{E}(d)} 1 - \frac{B}{n} =: r(d)$. Analogously to the considerations for structural selectivities, we can improve accuracy by considering covariance estimates. Then the total costs for the next batch of sorted accesses is shared by all Q candidates and incurs expected wasted cost:

$$EWC\text{-}SA \quad = \quad \sum_{d \in PQ} r(d) \cdot \frac{B \cdot m}{Q} \quad [SA].$$

We initiate the random accesses for tag-term score lookups and for structural conditions, if and only if $EWC\text{-}RA1(d) < EWC - SA\ and\ EWC\text{-}RA2(d) < EWC - SA$, respectively, with $RA's$ weighted 20 times higher than $SA's$. We actually perform the random accesses one at a time in ascending order of expected score gain (for tag-term score lookups) and selectivity (for path and twig conditions). Candidates that can no longer qualify for the top-k are eliminated as early as possible and their further random accesses are canceled.

7 Experiments

7.1 Setup

We implemented TopX as a collection of Java classes. We performed experiments on a 3 GHz dual Pentium PC with 2 GB of memory. Index lists were stored in an Oracle 10g server running on the same PC. Index lists were accesses in a multi-threaded mode and prefetched in batches of $B = 100$ tuples per index list and cached by the JDBC driver. The scan depth for invoking periodic queue rebuilds was synchronized with this prefetch size. The queue bound was set to $Q = 500$ candidates. We used three different datasets in the experiments.

The *INEX* collection [24] consists of full articles from IEEE Computer Society journals and conference proceedings in a rather complex XML format. We chose the 46 queries from the INEX topics of 2004 for which official relevance assessments are available. An example query is `//article[.//bibl[ "QBIC"] and .//p["image retrieval"]]`.

For the *IMDB* collection, we generated an XML document for each of the movies available at the Internet Movie Database (`www.imdb.com`). Such a document contains the movie's title, plot summaries, information about people such as name, date of birth, birth place, etc. The interesting issue in using this collection is the mixture between elements with rich text contents and categorical attributes such as `Genre=Thriller` yielding many ties in local scores. We asked colleagues to create 20 meaningful queries with structural and

keyword conditions; examples are queries of the kind `//movie[about(.//cast//casting//role, Sheriff)]//casting//actor[about(.//name, Henry Fonda)]`, thus looking for movies with Henry Fonda and an arbitrary actor in the role of a sheriff.

Since INEX and IMDB are still not exactly very large dataset in terms of documents, we included experiments on the TREC Terabyte corpus as a stress test, even if this is not XML. The *TREC Terabyte* collection of the TREC 2004 benchmark [39] consists of more than 25 million crawled Web pages from the .gov domain. This collection contains mostly HTML and PDF files. The 50 benchmark queries are mere keyword queries such as "train station security measures" or "aspirin cancer prevention".

Note that experimental studies in the literature on XPath, XQuery, and XML-IR system performance are mostly based on the XMark synthetic dataset (often using the 100 MB version which fits into memory), which is not really appropriate for our setting. INEX has become the main benchmark for XML IR. Table 1 shows the sizes of our test collections, Table 2 shows the disk resident sizes including all index structures required for sorted and random accesses described in Section 3.4. Index creation times were between 83 minutes for INEX and 15 hours for Terabyte, using standard IR techniques such as stemming, stop word removal, and the scoring model described in Section 3.2.

	#Docs	#El's	#Feat's	Size
INEX	12,223	12,071,272	119M	534MB
IMDB	386,529	34,669,538	130M	1,117MB
TB	25,150,527	n/a	2,938M	426GB

Table 1: Sizes of test collections

	INEX	IMDB	TB
Features	3.8GB	7.4GB	190.2GB
Elements	0.2GB	0.6GB	n/a
Histograms	7.4MB	11.5MB	13.5MB
Twigs&Paths	108KB	4KB	n/a
Total	**4.0GB**	**8.0GB**	**190.2GB**

Table 2: Index sizes for test collections

7.2 Competitors

Our experiments compared the following strategies for top-k query processing:

- *TopX-MinProbe* with the minimum probing strategy for random accesses, explained in Section 6.
- *TopX-BenProbe* with the cost-based strategy for beneficial random accesses, explained in Section 6.
- *TopX-Text*, using sorted access only to inverted term index lists for unstructured data. Here, random lookups are only used to clarify the final ranking among all top results' $[worstscore, bestscore]$ intervals.
- *Join&Sort*, a strategy where all query-relevant index lists are first completely fetched into main memory and the top-k algorithm works in memory using hash tables and cheap random access.
- *StructIndex*, the algorithm developed in [26] which uses a structure index to preselect candidates that satisfy the path conditions and then uses TA with random access to compute the top-k result.
- *StructIndex*$^+$, an optimized version of the StructIndex top-k algorithm, using the extent chaining technique of [26].

The *Join&Sort* competitor corresponds to a traditional DBMS-style query processing and is a lower bound for the amount of index list accesses any non-TA-based implementation would have to make. It is inspired by the Holistic Twig Join of [8, 25] which is probably the best known method for twig queries without any scoring or ranking (i.e., non-IR XPath).

The *StructIndex* competitor is driven by the assumption that structure conditions are often quite selective. It uses a Data-Guide-like structure index for first evaluating the structural skeleton of the query. This provides it with a compact representation of result candidates, namely, all element combinations that satisfy all path constraints, concisely encoded into combinations of "extent identifiers". These identifiers are stored as additional attributes in the entries of the inverted index lists; so we can quickly test, if an element that is encountered in the index scan for a tag-term condition belongs to a document that satisfies the structure conditions. As the algorithm also performs immediate lookups of missing tag-term scores (i.e., the original TA of Fagin [16]), it generally follows an eager strategy for random accesses.

The optimized *StructIndex*$^+$ method assumes that all elements in a tag-term index list that have the same extent identifier in the structure index, i.e., have the same path tags from the root to the elements, form a forward-linked chain. The evaluation of the structure conditions then groups the resulting candidate elements by extent identifier, and conceptually invokes one sorted index scan for each extent. The implementation merges these cursors into one index scan that is mostly sequential but can perform skips. The two *StructIndex* approaches have been optimized to fit our *full content* scoring. A comparison with Okapi BM25 scores for simple element contents has shown major benefits of our scoring model in terms of result quality. Furthermore, *StructIndex* also uses a hash-based cache structure to avoid redundant random accesses.

7.3 Baseline Runs

Table 3 gives detailed results comparing different TopX configurations and competitors for varying k for INEX, IMDB, and Terabyte. The table lists the following statistics each for the whole batch of queries: the parameter k, the amount of queries, the sum of sorted accesses and the sum of random accesses for the entire batch, the per-query average process CPU times in seconds, the sum of queue operations (inserts, deletes, and updates) for TopX, the maximum number of cached items for the hash-structure used (in numbers of documents), the macro-averaged precision at k (P@k) and the mean average precision (MAP) for the top-k using official relevance assessments for INEX and Terabyte. Since the IMDB is not an official benchmark setting, we omit the P@k and MAP values for the

IMDB, although results were very good throughout the whole range of queries. We do not report detailed wall-clock elapsed times, because we could not measure them with sufficient statistical confidence given that our testbed was not exclusively dedicated to a single process. Wall-clock times were typically a factor of 10 higher than CPU times (because of disk I/O by the Oracle server), yielding user-perceived response times in the range of 0.1 to 0.7 seconds per INEX query (and up to 30 seconds for Terabyte) at $k = 10$ and without probabilistic pruning, and this proportion was fairly consistent for all algorithms and parameter settings.

Table 3 shows that the conservative TopX method without probabilistic pruning ($\epsilon = 0$) reduces the number of expensive random accesses by a factor of 50 to 80 compared to the *StructIndex* competitors on INEX, and still by a factor of 4 to 6 on IMDB, with very good rates of inexpensive sorted accesses.

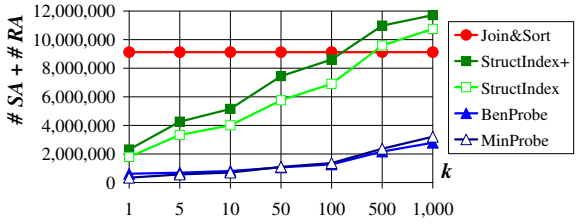

Figure 4: Performance as a function of k for INEX

The *MinProbe* scheduling outperforms *BenProbe* on INEX, in terms of saving random accesses. On IMDB, on the other hand, *BenProbe* was superior because of the different structural characteristics of the data. There are 106,970 distinct twig and 3,404 distinct path structures in INEX compared to only 3,859 distinct twigs and 111 distinct path structures in IMDB. Because of the low document-level selectivity of path conditions on IMDB, the structural constraint tests using range scans on the `Elements` table to get the pre/post codings were much more expensive than for INEX so that random access scheduling became very crucial. Here the cost-based *BenProbe* method showed its benefits. For the same reason, the *StructIndex* techniques became competitive to TopX for large k, but for small k up to 100 both *MinProbe* and *BenProbe* won by a large margin. Generally, sorted access shows its drawbacks for categorical attributes such as genres, birth places, etc., which yield very long lists with many ties in the local scores. Note that the sorted-access cost of *Join&Sort* is the same for all k values as the query-relevant index lists are completely fetched anyway (even for k=1). We report the MAP value (which depends on k) for the top $k = 1,000$ in this case.

For k as large as 1,000 or higher, all top-k approaches degenerate and lose their advantage over the database-style Join&Sort approach due to their overhead. But this is not surprising; very large k values are no longer a case for top-k algorithms at all. Note that for $k = 1,000$, we already return about 8% precent of the INEX documents as query results. For $k = 1,000$, *StructIndex* approaches has more index accesses than *Join&Sort* because of the different index structures, where TopX still has fewer index accesses than *Join&Sort*.

The run *Join&Sort-nr* for INEX using a non-redundant scoring model with Okapi BM25 weights on per-document statistics performs very poorly in terms of precision and MAP values for the top 1,000 results. This demonstrates the severe shortcomings of standard document scoring approaches for XML retrieval. Note that MAP captures both precision and recall and is the key metric in the relevance assessment in both benchmarks, INEX and TREC.

Terabyte served as a stress test to our engine. The relatively high P@1,000 values indicate that the relevance sets are huge as well. Therefore the MAP values at the lower k mostly suffer from not returning the top 10,000 as originally proposed by TREC 2004 which we do not consider meaningful for a top-k engine. For Terabyte we did not run *Join&Sort* but merely report the lower bounds of index access costs based on the relevant index lists' sizes.

7.4 Runs with Probabilistic Pruning

We also studied the influence of ϵ on performance and query result quality. The results for the *MinProbe* scheduling are shown in Table 4. As an additional quality measure we report the macro-averaged relative precision $rPrec(R_1, R_2) := \frac{|R_1 \cap R_2|}{\max\{|R_1|,|R_2|\}}$ compared to the conservative algorithm with $\epsilon = 0$.

The probabilistic pruning reduces both the amount of index accesses and the overhead in queue operations, whereas the predictor overhead itself is almost negligible. The performance gain is another factor of 20 in access rates and a factor of 10 in run times compared to the conservative TopX and more than two orders of magnitude compared to *StructIndex* or *Join&Sort* throughout all the collections, at very high precision values. Figure 5 shows performance gains for INEX, in terms of accesses rates, as a function of the ϵ value. Although there are minor reductions in the user-perceived quality measures like precision and MAP, probabilistic pruning hardly affects the result quality. Figure 6 shows that the relative precision value *rPrec* degrades at a much higher rate. This means that different results are returned at the top ranks, but they are equally good from a user perspective based on the official relevance assessments of INEX and TREC.

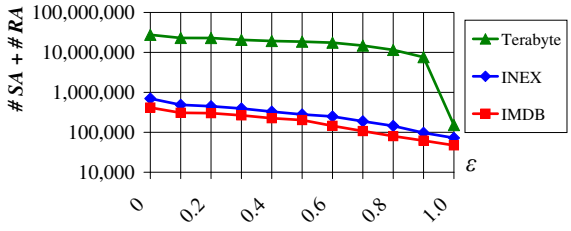

Figure 5: Performance as a function of ϵ

8 Conclusion

The TopX engine combines an extended TA-style algorithm with a focus on inexpensive sorted accesses with

Run	k	Queries	Sum(SA)	Sum(RA)	Avg(CPU)	Sum(Q.Ops)	Max(Cache)	P@k	MAP
INEX									
Join&Sort	1,000	46	9,122,318	0	0.260	n/a	12,180	0.03	0.17
Join&Sort-nr	1,000	46	8,189,566	0	0.245	n/a	12,183	0.01	0.07
StructIndex	1	46	289,160	1,500,804	0.161	n/a	2,725	0.57	0.04
	10	46	761,970	3,245,068	0.357	n/a	8,387	0.34	0.09
	100	46	1,966,960	4,938,645	0.648	n/a	10,415	0.13	0.15
	1,000	46	4,442,806	6,307,770	1.123	n/a	11,542	0.03	0.17
StructIndex$^+$	1	46	30,309	2,282,280	0.896	n/a	2,725	0.57	0.04
	10	46	77,482	5,074,384	1.837	n/a	8,387	0.34	0.09
	100	46	160,816	8,447,310	2.753	n/a	10,415	0.13	0.15
	1,000	46	271,803	11,441,431	3.658	n/a	11,542	0.03	0.17
TopX *BenProbe*	1	46	605,975	10,668	0.064	66,582	11,500	0.57	0.04
	10	46	723,169	84,424	0.078	71,959	11,640	0.34	0.09
	100	46	826,458	441,563	0.124	96,176	11,644	0.13	0.15
	1,000	46	882,929	1,902,427	0.352	98,692	11,683	0.03	0.17
TopX *MinProbe*	1	46	322,109	15,876	0.016	36,820	9,157	0.57	0.04
	10	46	635,507	64,807	0.026	59,457	9,828	0.34	0.09
	100	46	999,608	361,706	0.062	89,008	10,842	0.13	0.15
	1,000	46	1,219,639	1,984,801	0.260	112,423	11,058	0.03	0.17
IMDB									
Join&Sort	1,000	20	14,510,077	0	37.758	n/a	282,158	n/a	n/a
StructIndex	1	20	251,036	205,775	0.126	n/a	4,162	n/a	n/a
	10	20	346,697	291,655	0.163	n/a	5,071	n/a	n/a
	100	20	629,574	747,737	0,378	n/a	18,111	n/a	n/a
	1,000	20	1,274,624	1,735,399	1.006	n/a	23,831	n/a	n/a
StructIndex$^+$	1	20	15,250	208,140	0.153	n/a	4,162	n/a	n/a
	10	20	22,445	301,647	0.173	n/a	5,071	n/a	n/a
	100	20	67,065	782,856	0.437	n/a	18,111	n/a	n/a
	1,000	20	180,701	1,914,181	1.097	n/a	23,831	n/a	n/a
TopX *BenProbe*	1	20	202,429	8,672	0.066	27,663	24,534	n/a	n/a
	10	20	241,471	50,016	0.080	27,818	25,012	n/a	n/a
	100	20	248,080	187,684	0.126	29,351	23,825	n/a	n/a
	1,000	20	400,142	1,231,516	0.598	89,903	28,051	n/a	n/a
TopX *MinProbe*	1	20	181,973	13,889	0.031	36,245	3,196	n/a	n/a
	10	20	317,380	72,196	0.061	119,899	5,231	n/a	n/a
	100	20	870,615	241,955	0.349	156,895	21,374	n/a	n/a
	1,000	20	993,751	1,326,999	1.214	250,501	49,965	n/a	n/a
Terabyte									
Join&Sort	1,000	50	105,806,358	0	n/a	n/a	n/a	0.08	0.13
TopX *Text*	1	50	13,452,578	1,390	1.829	2,077,896	776,977	0.16	0.01
	10	50	27,541,711	2,035	19.840	5,540,426	1,661,793	0.31	0.01
	100	50	53,000,119	3,192	97.583	7,884,308	2,077,187	0.21	0.07
	1,000	50	56,619,220	25,227	33.015	7,113,318	2,381,891	0.08	0.13

Table 3: Baseline runs for INEX, IMDB, and Terabyte for various k

Run	ϵ	Queries	Sum(SA)	Sum(RA)	Avg(CPU)	Sum(Q.Ops)	Max(Cache)	P@10	MAP	rPrec
INEX										
TopX *MinProbe*	0.10	46	426,986	59,414	0.063	39,865	9,828	0.32	0.08	0.80
	0.25	46	392,395	56,952	0.056	38,568	9,828	0.34	0.08	0.77
	0.50	46	231,109	48,963	0.027	20,567	7,205	0.31	0.08	0.65
	0.75	46	102,118	42,174	0.017	7,223	3,167	0.33	0.08	0.51
	1.00	46	36,936	35,327	0.007	902	570	0.30	0.07	0.38
IMDB										
TopX *MinProbe*	0.10	20	250,173	57,066	0.028	106,166	5,231	n/a	n/a	0.95
	0.25	20	234,248	67,015	0.033	88,302	5,231	n/a	n/a	0.89
	0.50	20	147,471	55,197	0.026	47,027	5,231	n/a	n/a	0.80
	0.75	20	38,679	41,504	0.019	10,590	3,586	n/a	n/a	0.77
	1.00	20	10,068	37,058	0.013	663	568	n/a	n/a	0.78
Terabyte										
TopX *Text*	0.10	50	23,010,990	1,106	4.312	4,947,328	1,662,197	0.27	0.01	0.73
	0.25	50	22,948,882	871	9.952	4,884,704	1,661,793	0.27	0.01	0.67
	0.50	50	19,283,550	993	3.219	3,300,383	1,661,793	0.26	0.01	0.60
	0.75	50	11,538,263	641	1.362	2,223,093	1,661,793	0.22	0.01	0.51
	1.00	50	151,043	779	0.011	5,953	4,998	0.19	0.01	0.35

Table 4: TopX runs with probabilistic pruning for various ϵ at $k = 10$

a number of novel features: carefully designed, precomputed index tables and a cost-model for scheduling that helps avoiding or postponing random accesses; a highly tuned method for index scans and priority queue management; and probabilistic score predictors for early candidate pruning. Our performance experiments demonstrate the efficiency and practical viability of TopX for ranked retrieval of XML data.

In addition to the XML IR techniques presented in this paper, TopX efficiently integrates ontology/thesaurus-based query expansion, relaxing, for example, a search for the tag `<book>` to find also documents with tag `<monography>` in a statistically focused and efficiently computed manner. Other features include efficient phrase matching and the flexibility of handling query conditions either in a score-aggregating "andish" mode or as hard conjunctions with mandatory terms, tags, and paths. Our future work will generally aim to combine the best methods from structured XML querying and non-schematic IR,

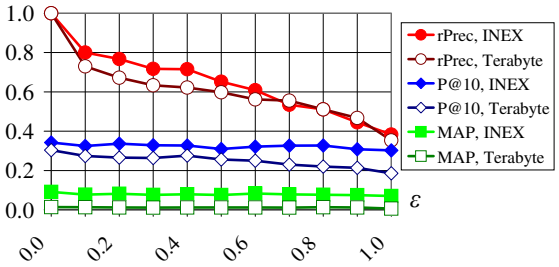

Figure 6: Precision as a function of ϵ

and will mostly address more sophisticated techniques for scheduling accesses to different kinds of content and structure indexes.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *VLDB*, pages 591–600, 2001.

[2] S. Agrawal et al. Automated ranking of database query results. In *CIDR 2003*, 2003.

[3] S. Al-Khalifa et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE 2002*, pages 141–152, 2002.

[4] S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying structured text in an XML database. In *SIGMOD 2003*, pages 4–15, 2003.

[5] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible structure and full-text querying for XML. In *SIGMOD 2004*, pages 83–94, 2004.

[6] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR*, pages 35–42, 2001.

[7] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE 2002*, pages 369–380, 2002.

[8] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD 2002*, pages 310–321, 2002.

[9] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *SIGIR*, pages 97–110, 1985.

[10] M. J. Carey and D. Kossmann. Reducing the braking distance of an sql query engine. In *VLDB 1998*, pages 158–169, 1998.

[11] D. Carmel et al. Searching XML documents via XML fragments. In *SIGIR 2003*, pages 151–158, 2003.

[12] K.-C. Chang and S.-W. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD 2002*, pages 346–357, 2002.

[13] T. T. Chinenyanga and N. Kushmerick. Expressive retrieval from XML documents. In *SIGIR 2001*, pages 163–171, 2001.

[14] S. Cohen et al. XSEarch: A semantic search engine for XML. In *VLDB 2003*, pages 45–56, 2003.

[15] A. P. de Vries, N. Mamoulis, N. Nes, and M. L. Kersten. Efficient k-nn search on vertically decomposed data. In *SIGMOD Conference*, pages 322–333, 2002.

[16] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[17] N. Fuhr and K. Großjohann. XIRQL: A query language for information retrieval in XML documents. In *SIGIR 2001*, pages 172–180, 2001.

[18] T. Grabs and H.-J. Schek. Powerdb-xml: Scalable xml processing with a database cluster. In *Intelligent Search on XML Data*, pages 193–206, 2003.

[19] T. Grust. Accelerating XPath location steps. In *SIGMOD 2002*, pages 109–120, 2002.

[20] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB 2000*, pages 419–428, 2000.

[21] L. Guo et al. XRANK: Ranked keyword search over XML documents. In *SIGMOD 2003*, pages 16–27, 2003.

[22] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE 2003*, pages 367–378, 2003.

[23] I. F. Ilyas et al. Rank-aware query optimization. In *SIGMOD 2004*, pages 203–214, 2004.

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

[25] H. Jiang et al. Holistic twig joins on indexed XML documents. In *VLDB 2003*, pages 273–284, 2003.

[26] R. Kaushik et al. On the integration of structure indexes and inverted lists. In *SIGMOD 2004*, pages 779–790, 2004.

[27] Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB 2004*, pages 72–83, 2004.

[28] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr. Xpathlearner: An on-line self-tuning markov histogram for xml path selectivity estimation. In *VLDB*, pages 442–453, 2002.

[29] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *VLDB*, pages 129–140, 2003.

[30] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM TODS*, 29(4):319–362, 2004.

[31] A. Marian et al. Adaptive processing of top-k queries in XML. In *ICDE 2005*, 2005.

[32] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE 1999*, pages 22–29, 1999.

[33] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *JASIS*, 47(10):749–764, 1996.

[34] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR*, pages 232–241, 1994.

[35] T. Schlieder and H. Meuss. Querying and Ranking XML Documents. *J. of the Am. Soc. for Information Science and Technology*, 53(6):489–503, 2002.

[36] A. Theobald and G. Weikum. Adding relevance to XML. In *WebDB 2000*, pages 105–124, 2001.

[37] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In *EDBT 2002*, pages 477–495, 2002.

[38] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB 2004*, pages 648–659, 2004.

[39] Text retrieval conference. `http://trec.nist.gov/`.

[40] A. Trotman and B. Sigurbjörnsson. Narrowed Extended XPath I (NEXI). available at `http://www.cs.otago.ac.nz/postgrads/andrew/2004-4.pdf`, 2004.

[41] C. Zhang et al. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD 2001*, pages 425–436, 2001.