The VLDB Journal manuscript No.
(will be inserted by the editor)

TopX

Efficient and Versatile Top-*k* Query Processing for Semistructured Data

Martin Theobald · Holger Bast · Debapriyo Majumdar · Ralf Schenkel · Gerhard Weikum

Received: date / Revised: date / Accepted: 11-06-2007

Abstract Recent IR extensions to XML query languages such as Xpath 1.0 Full-Text or the NEXI query language of the INEX benchmark series reflect the emerging interest in IR-style ranked retrieval over semistructured data. TopX is a top-*k* retrieval engine for text and semistructured data. It terminates query execution as soon as it can safely determine the *k* top-ranked result elements according to a monotonic score aggregation function with respect to a multidimensional query. It efficiently supports vague search on both content- and structure-oriented query conditions for dynamic query relaxation with controllable influence on the result ranking. The main contributions of this paper unfold into four main points: 1) fully implemented models and algorithms for ranked XML retrieval with XPath Full-Text functionality, 2) efficient and effective top-*k* query processing for semistructured data, 3) support for integrating thesauri and ontologies with statistically quantified relationships among concepts, leveraged for word-sense disambiguation and query expansion, and 4) a comprehensive description of the TopX system, with performance experiments on large-scale corpora like TREC Terabyte and INEX Wikipedia.

Keywords Efficient XML full-text search · content- and structure-aware ranking · top-*k* query processing · cost-based index access scheduling · probabilistic candidate pruning · dynamic query expansion · DB&IR integration

Martin Theobald · Holger Bast · Debapriyo Majumdar · Ralf Schenkel · Gerhard Weikum
Max-Planck Institute for Informatics, Saarbrücken, Germany
E-mail: {mtb,bast,dmajumda,schenkel,weikum}@mpi-inf.mpg.de

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, queries either return too many or too few results. Rather the ranked-retrieval paradigm is called for, with relaxable search conditions, various forms of similarity predicates on tags and contents, and quantitative relevance scoring.

TopX [91,92] is a search engine for ranked retrieval of XML data. It supports a probabilistic-IR scoring model for full-text content conditions and tag-term combinations, path conditions for all XPath axes as exact or relaxable constraints, and ontology-based relaxation of terms and tag names as similarity conditions for ranked retrieval. While much of the TopX functionality was already supported in our earlier work on the XXL system [87,88], TopX has an improved scoring model for better precision and recall, and a radically different architecture which makes it much more efficient and scalable. TopX has been stress-tested and experimentally evaluated on a variety of datasets including the TREC [94] Terabyte benchmark and the INEX [56] XML information retrieval benchmark on an XML version of the Wikipedia encyclopedia. For the INEX 2006 benchmark, TopX served as the official reference engine for topic development and some of the benchmarking tasks.

Research on applying IR techniques to XML data has started about five years ago [28,43,83,87] and has meanwhile gained considerable attention (see [8,12,32] and the references given there). 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 [56] as follows:

```
//book[about(., Information Retrieval XML)
      and about(.//reference, PageRank)]
   //author[about(.//affiliation, Stanford)]
```

This twig query should find the *best matches* for authors of books that contain the terms "Information Retrieval XML" and have descendants tagged as reference and affiliation with content terms "PageRank" and "Stanford", respectively. However, beyond such exact-match results, it should also find books with similar content, like books about "statistical language models for semistructured data", but possibly ranked lower than exact matches, and if no author from Stanford qualifies it may even provide books from someone at Berkeley as an approximate, still relevant result. In addition, as an additional feature (not expressible in NEXI), we may consider relaxing tag names so that, for example, monographs or even survey articles are found, too.

The challenge addressed in this paper is to process such queries with a rich mixture of structural and content-related conditions *efficiently*. The method of choice for top-k similarity queries is the family of threshold algorithms, developed by [40,49,73] and related to various methods for processing index lists in IR [15,23,74,13]. 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.

Applying this algorithmic paradigm to XML ranked retrieval is all but straightforward. 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 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" in bibliographic data sets should be viewed as specific when occurring within elements of type `<section>` or `<caption>` but is considered less informative in `<journalname>`.
- Relevant intermediate results of the 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.
- It should be possible to relax search terms and, in particular, tag names, using ontology- or thesaurus-based similarities. For example, a query for a `<book>` about "XML" should also consider a `<monograph>` on "semistructured data" as a result candidate. but such a query expansion should avoid using similarity thresholds that are difficult to tune manually, and it must be careful to avoid topic dilution that could result from over-expansion.

Thus, 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 disk-resident 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 System Overview

TopX aims to bridge the fields of database systems (DB) and information retrieval (IR). From a DB viewpoint, it provides an efficient algorithmic basis for top-k query processing over multidimensional datasets, ranging from structured data such as product catalogs (e.g., bookstores, real estate, movies, etc.) to unstructured text documents (with keywords or stemmed terms defining the feature space) and semistructured XML data in between. From an IR viewpoint, TopX provides ranked retrieval based on a scoring function, with support for flexible combinations of mandatory (conjunctive) and optional ("andish") conditions as well as advanced text predicates such as phrases, negations, etc. The key point, however, is that TopX combines these two viewpoints into a unified framework and software system, with emphasis on XML ranked retrieval.

Figure 1 depicts the main components of the TopX system. Software components are shown as light-grey rectangles; the numbered components are outlined in the following. TopX supports three kinds of front-ends: as a servlet

with an HTML end-user interface, as a Web Service with a SOAP interface, and as a Java API. It uses a relational database engine as a storage system; the current implementation uses Oracle10g, but the JDBC interface would easily allow other relational backends, too. The various TopX components fall into two categories: data-entry components and query-time components.

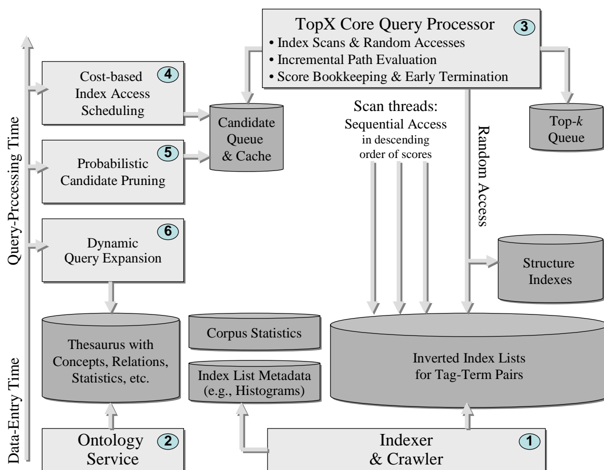

Fig. 1 TopX components.

At data-entry time, when new documents are entered, the *Indexer (1)* parses and analyzes the data, and builds or updates the index structures for efficient lookups of tags, content terms, phrases, structural patterns, etc. When dealing with Web or intranet or desktop data that has href hyperlinks, TopX can use its built-in *Crawler (1)* to traverse entire graphs and gather documents. An offline *Ontology Service (2)* component manages optional thesauri or lightweight ontologies with various kinds of semantic relationships among concepts and statistical weighting of relationship strengths. This component makes use of WordNet [42] and other knowledge sources such as Wikipedia.

At query-processing time, the *Query Processor (3)* decomposes queries and invokes the top-k algorithms based on index scans. It is in charge of maintaining intermediate top-k results and candidate items in a priority queue, and it schedules the sequential and random accesses on the precomputed index lists in a multi-threaded architecture. The Query Processor can make use of several advanced components that can be plugged in on demand and provide means for run-time acceleration:

- The *Index Access Scheduler (4)* provides a suite of scheduling strategies for sorted and random accesses to index entries. This includes simple heuristics that are reasonably effective and have very low overhead as well as advanced strategies based on probabilistic cost models that are even better in terms of reducing index-access costs but incur some overhead.
- The *Probabilistic Candidate Pruning (5)* component is based on mathematical models for predicting scores of candidates (based on histogram convolution and correlation estimates) and also for selectivity estimation (based on XML tag-term and twig statistics). This allows TopX to drop candidates that are unlikely to qualify for the top-k results at an early stage, with a controllable risk and probabilistic result guarantees. It gives the system a very effective way of garbage collection on its priority queue and other in-memory data structures.
- The *Dynamic Query Expansion (6)* component maps the query keywords to concepts in the available thesaurus or ontology and incrementally generates query expansion candidates. This is interleaved with the actual index-based query processing, and provides TopX with an efficient and robust expansion technique for both content terms and XML tag names (i.e., element or attribute names).

1.3 Contribution and Outline

This paper provides a comprehensive view of the complete TopX system, integrating its various technical components. The paper is based on but significantly extends earlier conference papers that relate to TopX, namely [93] on probabilistic methods for efficient top-k queries, [90] on efficient query expansion, [91] on index-based query processing for semistructured data in TopX, and [18] on index-access scheduling. More specifically, the current paper makes the following value-added contributions that extend our own prior work: a detailed description of our scoring model for ranked retrieval based on an XML-specific extension of the probabilistic-IR Okapi BM25 model [79], extended techniques for efficient indexing and query processing based on hybrid forms of tree-encoding [47,48] and data-guide-like methods [44,60], a detailed description of integrating thesauri and ontologies, and experimental studies on the Wikipedia XML collection of the INEX 2006 benchmark. Overall, the research centered around TopX makes the following major contributions:

- comprehensive support for the ranked retrieval functionality of XPath Full-Text [100], including a probabilistic-IR scoring model for full-text content conditions and tag-term combinations, path conditions for all XPath axes as exact or relaxable constraints, and ontology-based relaxation of terms and tag names as similarity conditions for ranked retrieval,
- efficient and scalable techniques for content-and-structure indexing and query processing, with demonstrated good performance on large-scale benchmarks,

- probabilistic models for approximate top-*k* query processing that predict scores in sequential index scans and can thus accelerate queries by earlier termination and lower memory consumption,
- judicious scheduling of sequential and random index accesses for further run-time improvements, specifically designed for handling XML data, and
- efficient support for integrating ontologies and thesauri, by incremental merging of index lists for on-demand, self-throttling query expansion.

The complete TopX system is available as open source code from the URL `http://topx.sourceforge.net`.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents the query-language functionality. Section 4 presents the scoring model for ranked retrieval. Section 5 introduces the indexes and the query processor of TopX (components (1) and (3) of Figure 1). Sections 6 and 7 discuss techniques that improve efficiency: Section 6 is on scheduling strategies for random accesses for testing expensive predicates such as structural path conditions (component (4)), and Section 7 is on probabilistic pruning of top-*k* candidates (component (5)). Section 8 presents the integrated support for thesauri and ontologies and their efficient use in query expansion (components (2) and (6)). Section 9 extends the basic indexing scheme by hybrid indexes for speeding up particular kinds of queries (extension of component (1)). Section 10 discusses implementation issues. Section 11 presents our performance experiments.

2 Related Work

2.1 IR on XML Data

Efficient evaluation and ranking of XML path conditions is a very fruitful research area. Solutions include structural joins [3], the multi-predicate merge join [103], the Staircase join based on index structures with pre- and postorder encodings of elements within document trees [47] and Holistic Twig Joins [22,59]. The latter, aka. path stack algorithm, is probably the most efficient method [29] for twig queries using sequential scans of index lists and linked stacks in memory. However, it does not deal with uncertain structure and does not support ranked retrieval or top-*k*-style threshold-based early termination.

Vagena et al. [96] apply structural summaries to efficiently evaluate twig queries on graph-structured data, and Polyzotis et al. [75] present an efficient algorithm for computing (structurally) approximate answers for twig queries. [64] extends XQuery to support partial knowledge of the schema. None of these papers considers result ranking and query optimization for retrieving the top-*k* results only.

Information retrieval on XML data has become popular in recent years; [12] gives a comprehensive overview of the field. Some approaches extend traditional keyword-style querying to XML data [31,51,54]. [25,28,43,87] introduced full-fledged XML query languages with rich IR models for ranked retrieval. [25,45,70] developed extensions of the vector space model for keyword search on XML documents, whereas [66] use language models for this purpose. [67,83] addressed vague structural conditions, [11] combined this theme with full-text conditions, and [10] proposed an integrated scoring model for content and vague structural conditions. More recently, various groups have started adding IR-style keyword conditions to existing XML query languages. TeXQuery [7] is the foundation for the W3C's official Full-Text extension to XPath 2.0 and XQuery [100]. [41] extends XQuery with ranking for keyword conditions and presents a pipelined architecture for evaluating queries but does not consider finding only the best results. [4] introduced a query algebra for XML queries that integrates IR-style query processing.

TIX [4] and TAX [58] are query algebras for XML that integrate IR-style query processing into a pipelined query evaluation engine. TAX comes with an efficient algorithm for computing structural joins. The results of a query are scored subtrees of the data; TAX provides a threshold operator that drops candidate results with low scores from the result set. TOSS [55] is an extension of TAX that integrates ontological similarities into the TAX algebra. XFT [9] is an algebra for complex full-text predicates on XML that comes together with an efficient evaluation algorithm; it can be integrated with algebras for structured XML search such as TIX.

Recent work on making XML ranked retrieval more efficient has been carried out by [61] and [68]. [61] uses path index operations as basic steps; these are invoked within a TA-style top-*k* algorithm with eager random access to inverted index structures. The scoring model can incorporate distance-based scores, but the experiments in the paper are limited to DB-style queries over a synthetic dataset rather than XML IR in the style of the INEX benchmark [56], which is using a large annotated collection of IEEE Computer Society publications (or, lately, an annotated version of Wikipedia).

[68] focuses on the efficient evaluation of approximate structural matches along the lines of [7]. It provides different query plans and can switch the current query plan at runtime (i.e., the join order of individual tuples following ideas of [16]) to speed up the computation of the top-*k* results. The paper considers primarily structural similarity by means of outer joins but disregards optimizations for content search.

Our own prior work on ranked XML retrieval has been published in [18,93,90,91], we discussed the relationship of these papers to the current paper in Section 1.3.

2.2 Top-k Threshold Algorithms

The state of the art on top-k queries over large disk-resident (inverted) index lists has been defined by seminal work on variants of so-called Threshold Algorithms (TA) [37, 39, 40, 49, 50, 73]. Assuming that entries in an index list are sorted in descending order of scores, TA scans all query-relevant index lists in an interleaved manner and aims to compute "global" scores for the encountered data items by means of a monotonic score aggregation function such as (weighted) sum, or maximum, etc. The algorithm maintains the worst score among the current top-k results and the best possible score for all other candidates and items not yet encountered. The latter then serves as a threshold for stopping the index scans when no candidate can exceed the score of the currently k^{th} ranked result. The algorithm comes in three variants: 1) The *original TA* approach eagerly looks up all local scores of each encountered item and thus knows the full score immediately when it first encounters the item. 2) Since random accesses may be expensive and, depending on the application setting, sometimes infeasible, the alternative *No-Random-Access Algorithm (NRA)* (coined Stream-Combine in [50]) merely maintains such *worstscore* and *bestscore* bounds for data items based on partially computed aggregate scores and using a priority queue for candidate items. Its stopping test compares the *worstscore* of the kth ranked result (typically coined *min-k*) with the *bestscore* of all other candidates. 3) Hybrid approaches, such as the *Combined Algorithm (CA)* [38], extend NRA by a simple cost-model for a few carefully scheduled random accesses to resolve the final scores of the most promising candidate items.

Obviously, TA is more effective in pruning the index scans and, thus, typically stops after a lower number of overall index accesses than NRA; but NRA completely avoids expensive random accesses and, therefore, can potentially achieve better run-times. CA, finally, aims at minimizing the overall query cost with regard to an environment-specific cost ratio c_R/c_S of random versus sorted accesses and therefore is the most versatile approach for a wide range of system and middleware setups.

Numerous variants of the TA family have been studied for multimedia similarity search [27, 72, 98], ranking query results from structured databases [2], and distributed preference queries over heterogeneous Internet sources such as digital libraries, restaurant reviews, street finders, etc. [26, 69, 102]. Marian et al. [69] have particularly investigated how to deal with restrictive sources that do not allow sorted access to their index lists and with widely varying access costs. To this end, heuristic scheduling approaches have been developed, but the threshold condition for stopping the algorithm is a conservative TA-style test. The IR community has also discussed various algorithms that perform smart pruning of index entries in impact-sorted or frequency-sorted inverted lists [23, 71, 74, 13–15]; these algorithms are very much TA-style combined with heuristic and tuning elements. Other top-k query algorithms in the literature include nearest-neighbor search methods based on an R-tree-like multidimensional index [5, 20, 30, 52, 53] and mapping techniques onto multidimensional range queries [21] evaluated on traditional database indexes. In this context, probabilistic estimators for selecting "cutoff" values have been developed by [30, 36, 86] and applied to multidimensional nearest-neighbor queries.

3 Data Model, Query Language & Representation

This section defines our data model and presents the external and internal representations of queries in TopX. TopX supports queries according to both the highly expressive XPath 2.0 Full-Text specification [6] and the NEXI [95] language used in the INEX benchmark series, which is narrowing the usage of XPath axes to only the *descendant* (`//`) and the *self* (`.`) axes and introduces an IR-style `about` operator instead of `ftcontains` in XPath Full-Text. In the following, we will mostly refer to the simpler NEXI syntax, as it captures most of the expressiveness needed for IR-style vague search of XML data considered in this paper, and, as a special case, it allows the formulation of traditional keyword queries over XML elements. As we will see, we allow slight, XPath-like extensions for the path structure of the NEXI syntax.

3.1 Data Model

As for our data model, we focus on a tree model for semistructured data, thus following the W3C XML 1.0 and 1.1 specifications, but disregarding any kind of meta markup (`<!..>`) and links in the form of XLink or ID/IDRef attributes. Attributes are treated as children of the respective element nodes, whereas text nodes are directly associated with their preceding element parents. Section 4 provides full details on our special handling of text nodes. Figure 6 shows a very simple example XML document that conforms to our model. Currently, all index structures employed by TopX as well as our top-k-style query processing (see Section 5) rely on data trees; a generalization to arbitrary data graphs is subject of future work.

3.2 Query Model

Figure 2 shows an example query written in a NEXI-style syntax[1].

[1] Strictly speaking, this query is not valid NEXI, since it contains a location path of more than two steps, but it would be allowed, with different syntax, in XPath Full-Text, and is supported by TopX.

```
//article[//bib[about(.//item, W3C)]]
  //sec[about(.//title, XML retrieval)]
    //par[about(.,native XML databases)]
```

Fig. 2 NEXI-style example query.

According to both the XPath and NEXI specifications, the rightmost, top-level node test of a location path is called the *target element* of the query; all other node tests in the location path denote the query's *support elements*. That is, the target element of the query defines the result granularity, and in a strict interpretation of the query structure, only those elements that match the query's target element are considered to be valid results. In the example of Figure 2 the `par` element is the target element, and the nodes labeled `article`, `bib`, `item`, `sec`, and `title` are support elements.

Instead of an explicit tag name, a query may also specify a wildcard '*' that is matched by any tag. The special query that consists only of a tag wildcard and some content conditions like in

```
//*[about(.,native XML databases)]
```

corresponds to a keyword-only query; following the INEX notation, these queries are called *content-only (CO)* queries, as opposed to *content-and-structure (CAS)* queries that contain additional structural constraints. The example in Figure 2 is a CAS query.

Using the full XPath syntax including forward and backward axes, path queries form *directed graphs*, potentially having cycles. However, the TopX query processor is currently restricted to directed acyclic graphs (DAGs). Note that in NEXI, using only forward axes between location steps and the self axis only in `about` operators, the formulation of queries by the user is even restricted to trees through the syntax.

3.3 Internal Query Representation

The query interpreter analyzes the query and decomposes it into a number of *navigational* and *content conditions* that form the nodes of the query DAG. These nodes are connected through typed edges, which we will refer to as *structural constraints*, each of which corresponds to an XPath axis such as the descendant or the self axis. In the above example query, the occurrences of elements labeled `section` and `title` are navigational conditions, the required occurrences of the terms "XML" and "retrieval" in the latter element are content conditions, and the requirement that the `section` element is connected to the `title` element by the descendant axis is a structural constraint.

The engine's internal representation of the query is purely DAG-based and – after parsing the query – becomes independent of the query-language-specific syntax. Figure 3 shows such an internal representation for the example of Figure 2. Here, the leaf nodes capture content conditions of each `about` operator, all non-leaf nodes correspond to navigational conditions, and the edges capture structural conditions.

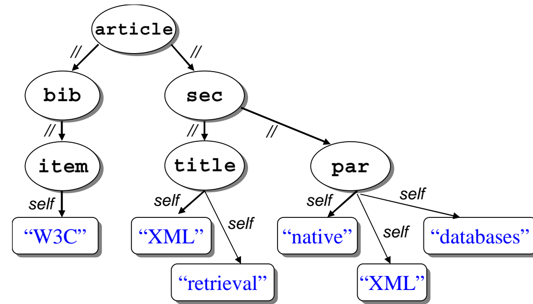

Fig. 3 Initial tree representation of the example NEXI query.

The main building blocks for the query processing are *tag-term pairs* obtained from merging the tokens in the `about` operators with their immediate parents in the path query. This also works in the DAG case, since for the content conditions, the last preceding navigational tag is always unique. As we will see later, there are very efficient ways of evaluating the tag condition and the term condition of such pairs together, and we therefore merge them into combined tag-term content conditions in a refined version of the query DAG.

Figure 4 shows the resulting structure for our example query of Figure 2. As this structure represents multiple term conditions for the same element in different query nodes, we now need to explicitly express that the query result must bind the leaves' parents with the same navigational condition (the parents' tag names in this case) to the very same element of a qualifying document. For example, the three nodes labeled `par` must be bound to the same result element. To capture this constraint, we connect all conditions that refer to the same element by a structural constraint edge that refers to the self axis (shown as dashed lines in Figure 4).

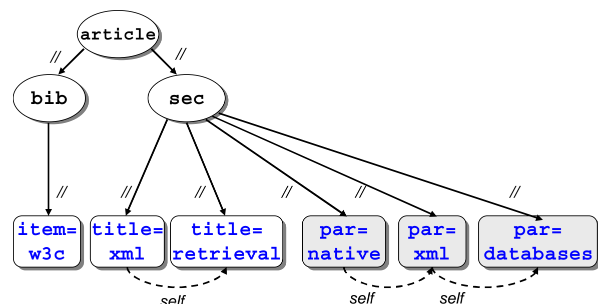

Fig. 4 DAG representation of the example query with combined tag-term conditions.

Note that the query representation of the original NEXI query tree has now become a DAG. As TopX supports all XPath axes and thus goes beyond NEXI, structural constraints that do not refer to the descendant axis would require further edges. For example, if the query had a condition that the `bib` element should follow the `sec` element, we would simply add an edge from the `sec` node to the `bib` node referring to the `following` axis of XPath. This query decomposition and internal DAG representation helps us to prepare the query for efficient evaluation by means of content and structure indexes.

To this end, it is helpful to consider also the transitive closure of descendant-axis edges in the query DAG. This is important when result candidates match some but not all of the structural constraints and we are willing to relax the structural skeleton of the query for an approximate result, for example, when a document has `title` and `par` elements that contain all the specified content terms but are not descendants of a `sec` element and there may not even be a `sec` element in the document.

The DAG representation can easily capture such transitive constraints, as shown in Figure 5 for the example query (where the transitive constraints are depicted as dotted lines). We will later see that we can now conveniently view all nodes and their outgoing edges of this transitively expanded query DAG as the *elementary query conditions* on which the query processor can operate. In slightly oversimplified terms, the goal of the query processor then is to find documents that match as many of these elementary query conditions as possible and with high scores. The actual scoring model will be explained in the subsequent section.

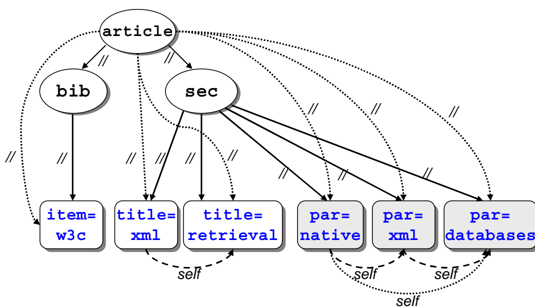

Fig. 5 Transitively expanded query DAG.

Orthogonally to the query formulation, TopX can be configured to return two different granularities as results: in *document mode*, TopX returns the best documents for a query, whereas in *element mode*, the best target elements are returned, which may include several distinct elements from the same document.

4 Relevance Scoring Model for XML Ranked Retrieval

In this section, we define our relevance scoring model that we use for ranked retrieval of XML data. The model captures the influence of content, navigational, and structural query conditions as defined in the previous section for IR-enhanced path queries written in the XPath Full-Text or NEXI syntax.

Recall that in element mode, only matches to the target element of the query are returned as results, which may themselves be part of larger subtrees embedded into the document tree. That is, only `par` elements may be returned by the query of Figure 2. In document mode (as demanded by some IR applications), we consider only the best of these subtrees in the document and return either the document root node or some user-defined *entry point* which may—but does not have to—be the target element of the query (see [77] for an IR discussion on how to determine the best entry points). Thus, our scoring model is based on the following building blocks (where each of the following subsections defines in detail what a “match” and its respective score is):

1) Content-related query conditions in `about` operators (or `ftcontains` in XPath Full-Text, respectively) are split into combined tag-term pairs. Each matched tag-term pair obtains a precomputed IR-style relevance score (Subsection 4.1).
2) XPath location steps are split into single node tests. Each navigational query condition that is not part of a tag-term pair contributes to the aggregated score of a matched subtree in the document by a static score mass c if all transitively expanded structural constraints rooted at it can be matched (Subsection 4.2).
3) In element mode, multiple valid embeddings of the query DAG into the document tree may be found for a each target element. In this case, we return for each target element e in document d the maximum score of all subtrees in d that match the query DAG and contain e (Subsection 4.3).
4) In document mode, we return for each document d the maximum score of matched target elements in d (Subsection 4.4).
5) In addition, Subsection 4.5 introduces IR-style extensions of this scoring model to support advanced search features like mandatory keywords, negations, and phrase matching inside `about` operators.

Processing combined tag-terms as our major building blocks for queries yields benefits for the scoring model as well as the query processing:

– Tags provide us with an initial context for refining the scoring weights of a given term. For example, in a corpus of IEEE journal papers, the tag-term pairs `par=transactions` and `bib=transactions` might relate to different meanings of transactions.

– Joint tag-term pairs tighten the processing through reduced query dimensionality and lower joint selectivity as compared to processing inverted lists for the respective tags and terms separately.

This model applies to both conjunctive query interpretations, where all query conditions must be matched, and more IR-style, so-called "*andish*" interpretations, where the result ranking is only determined through score aggregations, but some query conditions may not be matched at all.

4.1 Content Scores

We first define the partial score that an element e obtains when matched against a single `about` operator in the query. We define an element e (i.e., a node in an XML document tree) to satisfy a tag-term content condition if e matches the tag name, and the subtree rooted at e contains the term. We refer to all the terms in this subtree as the *full-content* of the element. More precisely, the $full\text{-}content(e)$ of element e is the concatenation of its own textual content and the textual contents of all its descendants (in document order if ordering is essential, e.g., for phrase matching).

The relevance of a tag-term match (e.g., derived from term-occurrence frequencies) influences the score of the matching element and its final ranking. More specifically, we make use of the following statistical measures that view the full-content of an element e with tag name A as a bag-of-words:

1) the *full-content term frequency*, $ftf(t,e)$, of term t in element e which is the number of occurrences of t in the full-content of e;
2) the *tag frequency*, N_A, of tag A which is the number of elements with tag name A in the entire corpus;
3) the *element frequency*, $ef_A(t)$, of term t with regard to tag A which is the number of elements with tag name A that contain t in their full-contents in the entire corpus.

Figure 6 depicts an XML example document, and Figure 7 illustrates our logical view of that document, with text nodes for each element using pre- and postorder labels [47] as node identifiers and for tree navigation. In the example, the full term frequency (*ftf*) of the term `xml` for the root element `article` has a value of 6, which reflects that the whole `article` element has a high probability of being relevant for a query containing the term `xml`. The tag frequency of `sec` elements is $N_{sec} = 2$, whereas the tag frequency of `article` elements is $N_{article} = 1$. The element frequency of the term `xml` in `sec` elements is $ef_{sec}(xml) = 2$. Figure 7 also shows (fictitious) content scores for some query-relevant elements and terms.

Now consider an elementary tag-term content condition of the form `A=t` where `A` is a tag name and t is a term that should occur in the full-content of an element[2]. The score of element e with tag name `A` for such a content condition should reflect:

```
<article id="ieee/w4043">
  <title>XML Data Management
    and Retrieval</title>
  <abs>XML data management systems vary
    widely in their expressive power.
  </abs>
  <sec>
  <st>Taking the Middle Ground</st>
  <par>XML management systems should perform
    well for both data-oriented and information-
    retrieval type queries.</par>
  </sec>
  <sec>
    <st>Native XML Databases</st>
    <par>Native XML databases
      can store schemaless data.</par>
  </sec>
  <bib>
   <item>
     XML Path Language (XPath) 1.0
     <url>www.w3c.org/TR/xpath</url>
   </item>
  </bib>
</article>
```

Fig. 6 Example XML document.

– the ftf value of the term t, thus reflecting the *occurrence statistics* of the term for the element's content,
– the *specificity* of the search term, with regard to tag-name-specific $ef_A(t)$ and N_A statistics for all element tags, and
– the *size* and, thus, compactness of the subtree rooted at e that contains the search term in its full-content.

Our scoring of element e with regard to condition `A=t` uses formulas of the following template:

$$score(e, \mathtt{A} = \mathtt{t}) = \frac{occurrence \cdot specificity}{size(e)}$$

Here, *occurrence* is captured by the ftf value, *specificity* is derived from the N_A and $ef_A(t)$ values, and *size* 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.

We could now specialize this formula into a simple *TF·IDF*-style measure, but 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 a high term frequency of a few

[2] Note that we now switch to this abbreviated notation for tokenized tag-term conditions which would conform to `A[about(., t)]` in the full NEXI syntax.

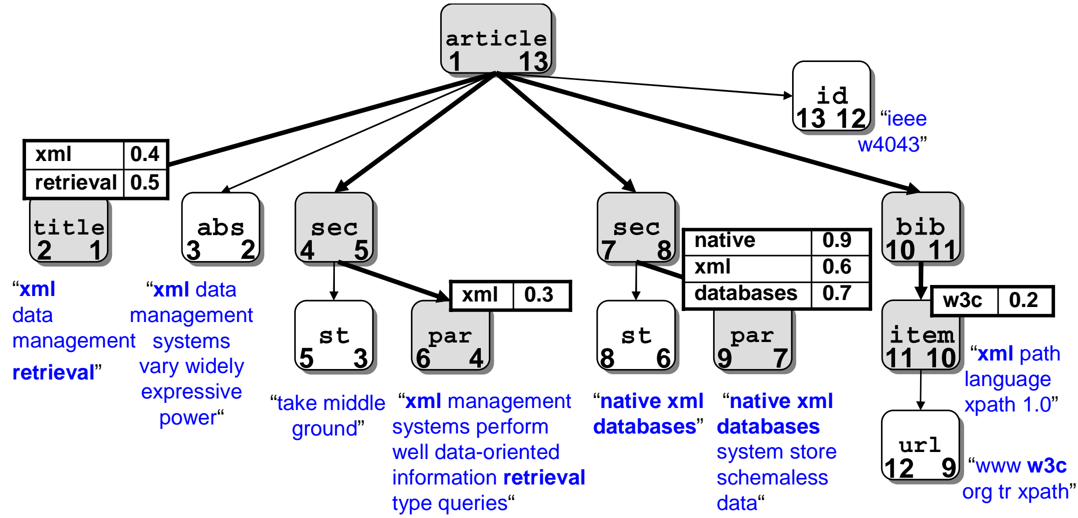

Fig. 7 Logical view of the document with some example full-content scores for the query-relevant elements.

rare terms. Likewise, the instantiation of compactness in the above formula should also use a dampened form of element size. To address these considerations, we have adopted the popular and empirically very successful Okapi BM25 scoring model (originating from probabilistic IR for text documents [79]) to our XML setting, thus leading to the following scoring function:

$$score(e, \mathtt{A} = \mathtt{t}) = \frac{(k_1+1)\, ftf(t,e)}{K + ftf(t,e)} \cdot \log\left(\frac{N_A - ef_A(t) + 0.5}{ef_A(t) + 0.5}\right) \quad (1)$$

$$\text{with } K = k_1\left((1-b) + b\frac{\sum_{s \in \text{ full content of } e} ftf(s,e)}{avg\{\sum_{s'} ftf(s',e') \mid e' \text{ with tag } \mathtt{A}\}}\right)$$

Note that the function includes the tunable parameters k_1 and b just like the original BM25 model. The modified function provides a dampened influence of the ftf and ef parts, as well as a compactness-based normalization that takes the average compactness of each element type into account.

For an `about` operator with multiple terms that is attached to an element e, the aggregated score of e is simply computed as the sum of the element's scores over the individual tag-term conditions, i.e.:

$$score(e,q) = score(e, \mathtt{A}[\mathtt{about}(., t_1, \ldots, t_m)) = \sum_{i=1}^{m} score(e, \mathtt{A} = \mathtt{t_i}) \quad (2)$$

Note that predicates of the form `about(.//A,` $t_1,\ldots,t_m$`)` and `A[about(.,` $t_1,\ldots,t_m$`)]` are treated equivalently in our setting (due to merging terms with their immediate parent tag).

4.2 Structural Scores

Our structural scoring model essentially counts the number of navigational (i.e., tag-only) query conditions that are satisfied by a result candidate and thus connect the content conditions matched for the different `about` operators. It assigns a small, constant, and tunable score mass c for every navigational condition that is matched and not itself part of a tag-term pair. Recall that every navigational condition corresponds to exactly one node in the query DAG. A navigational condition is matched by an element e in document d, if all the structural constraints, i.e., the element's outgoing edges, of the transitively expanded query DAG are satisfied.

To illustrate this approach, consider the example query of Figure 2 and its transitively expanded query DAG shown in Figure 5. The DAG has 14 descendant edges, some reflecting the 8 original descendant-axis conditions, some their transitive expansions; we do not consider the self axis edges between content conditions here. A structurally perfect result would match all 14 edges for the 3 non-leaf nodes of the query DAG, earning a structure score of $3 \cdot c$ for the navigational query conditions `article`, `bib`, and `sec`. When matching the structural constraints against the document tree in Figure 7, we see that our example document is only a near match to the structure of the query, since the `title` element matching the `title=XML` and `title= retrieval` conditions is in fact a sibling of the `sec` containing a `par` element that is matching the `par=native`, `par=XML`, and `par=databases` conditions, rather than a descendant as demanded by the query. Thus, any (partial) embedding of the query DAG into this document tree misses at least one structural score c for the unmatched navigational `sec` condition.

While our current use of the tunable parameter c is relatively crude (our experiments simply set $c = 1.0$ with con-

tent scores being normalized to ≤ 1.0 and thus put high emphasis on structural constraints), our scoring framework could be easily refined in various ways. We could introduce different c values for different types of structural constraints, for example, specific for the tag names of a query-DAG edge or specific for the axis that an edge captures (e.g., giving higher weight to descendant-axis edges than to following-axis or sibling-axis edges). Or we could even make c dependent on the goodness of an approximate structural match; for example, when a child-axis edge is not matched but the candidate result has a descendant-axis connection with matching tag names, we may still assign a relatively high c value (but lower than for a perfect child-axis match). Studying such extensions is left for future work.

4.3 Element Scores

In element mode, our algorithm returns a ranked list of target elements per document, using the target element as well as connected support elements of the query to aggregate scores from different navigational and content conditions that match the query DAG. For matches from multiple documents, these ranked lists of target elements are then merged to yield the final result ranking of target elements across documents.

We define $T(d)$ as the set of all elements in d that match the target element of the query; for example, any `par` element of d is a potential result of our example query. When aggregating scores for a target element $e \in T(d)$ from all elementary content conditions C (i.e., tag-term pairs) and navigational conditions N that are not part of a tag-term pair, we need to find valid embeddings of the query DAG into the document tree in the form of connected subtrees $Trees(e)$ that contain e. For each such subtree, the total score is the sum of the scores for all satisfied content and structure conditions. Since there may be multiple such embeddings for each target element, we define the score of a target element e to be the maximum aggregated score among all these embeddings $S \in Trees(e)$.

$$score(e,q) = \max_{S \in Trees(e)} \left\{ \sum_{e' \in S \cap C} score(e',q) + \sum_{e' \in S \cap N} c \right\} \quad (3)$$

Finding these subtree embeddings is a well studied problem in XPath query processing [3,22,47,59,103]. Details for our incremental, top-k-style XPath algorithm are discussed in Section 5.3.

In Figure 7, we have two matches $par[6,4]$ and $par[9,7]$ of target elements for our example query. The corresponding tree embeddings are shown in Figure 8. Here, $par[6,4]$ only matches the condition `par=xml` with a content score 0.3, and additionally aggregates content scores of 0.2, 0.4 and 0.5 for the matches to `item=w3c`, `title=xml` and `title=retrieval`, respectively. It additionally aggregates a structural score of $2 \cdot c$ for the matches of the two `article` and `bib` navigational conditions, as all transitive structural constraints rooted at these elements are satisfied. Setting $c = 1.0$, this yields a total score of 3.4 for $par[6,4]$. Similarly, the aggregated score of $par[9,7]$ is 0.9+0.6+0.7+0.2+0.4+0.5+1 +1=5.3, and probably yields a better result for the query.

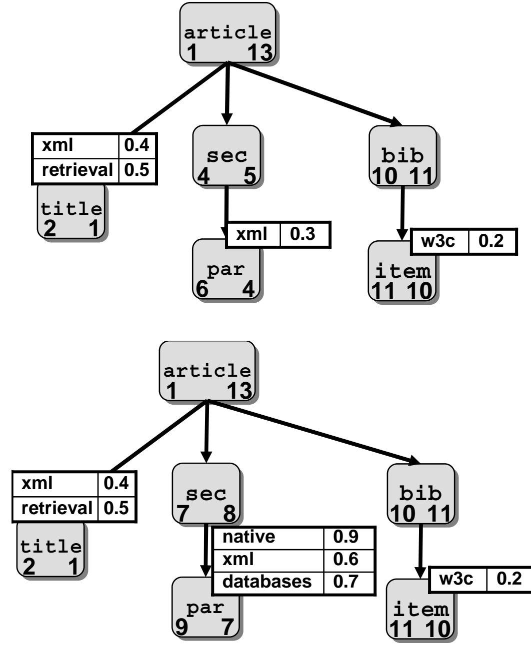

Fig. 8 Two embeddings of the query DAG from Figure 5 into the document tree of Figure 7.

Note that the value of the tunable constant c determines whether we favor matching the query structure or the content conditions in the non-conjunctive (i.e., "andish") mode. A large value of c tends to dominate the content conditions and will make the algorithm choose results that match the support elements of the query and neglect lower-scored content conditions that do not match the structure. A low value of c, on the other hand, tends to favor the content conditions and might still accept some support elements remaining unmatched among the top-ranked results. As an example for this, assume that our example document contains another `item` element with a high-scoring match for `item=w3c`, but connected to a `list` parent. In this situation, there would be another possible embedding that contains $par[9,7]$, which is shown in Figure 9. Using the default value $c = 1.0$, this embedding would get a total score of 4.9 as it earns only one structural score c (for the `article` match), even though its content score is better than in the embedding shown in Figure 8. If we set $c = 0.2$, the embedding with the `list` element gets a total score of 4.1 because of its high-scoring content match, which is higher than the score 3.7 of the other

embedding. In conjunctive mode, on the other hand, when all query conditions have to be matched by all valid results, tuning c would indeed affect the absolute scores but not the ranking among results.

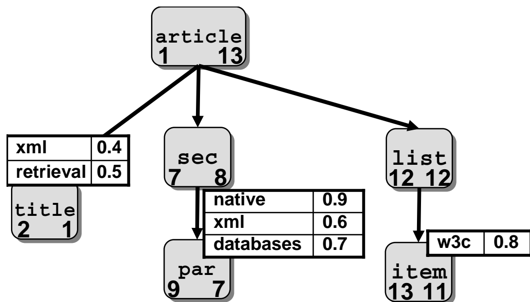

Fig. 9 Additional embedding in a slightly changed document.

4.4 Document Scores

In document mode, every document d inherits the score of its highest-scoring target element $e \in T(d)$, and these document scores determine the output ranking among documents.

$$score(d,q) = \max_{e \in T(d)} \{score(e,q)\} \tag{4}$$

For our example document and query this is $p[9,7]$ with a score of 5.3.

For an andish query evaluation with only partial knowledge on a document's content and structure, $score(d,q)$ should be a lower bound for the final score of the document at any time of the evaluation, and this score should monotonically increase as we gain more knowledge about both the content and structure of a result candidate. Efficient algorithms for this type of processing are discussed in Section 5.

4.5 IR Extensions for Advanced Query Features

4.5.1 Mandatory Terms

Unlike in a database-style boolean query, terms marked as mandatory reflect the user's intention that the terms will most likely occur in relevant documents, but there may as well be some relevant documents that do not contain the terms. Hence such conditions should not be evaluated as strict boolean filters, but instead boost the score of a document that satisfies them.

Formally, let $M \subseteq \{1,\ldots,m\}$ be a set of content conditions marked by a '$+$', to denote that the corresponding terms must occur in results. Then the aggregated score of a candidate element e of document d is defined as

$$score(e,t_i) = \begin{cases} (\beta_i + s_i(e)) & for\ d \in L_i \\ 0 & otherwise \end{cases} \tag{5}$$

with $\beta_i = 1$ for $i \in M$ and $\beta_i = 0$ otherwise. This way, mandatory terms are boosted by an additional boost factor β_i for elements that occur in the inverted list for condition i. If β_i is chosen reasonably large (again in an IR-style notion of vague search), e.g., $\beta_i = 1$ for $i \in M$, elements that match the mandatory conditions are very likely to be among the top matches regardless of their actual local scores $s_i(e)$.

4.5.2 Negations

The semantics of negations for non-conjunctive (i.e., andish) query evaluation is all but trivial. To cite the authors of the NEXI specification [95], "the user would be surprised if a '$-$' word is found among the retrieved results". This leaves some leeway for interpretation and commonly leads to the conclusion that the negated term should merely lead to a certain score penalty and should not completely eliminate all documents containing one of the negated tag-term pairs like in a conjunctive setup. Thus, for higher recall, a match to a negated query condition does not necessarily render the result irrelevant, if good matches to other content-related query conditions are detected.

In contrast to mandatory search conditions, the scoring of negated tag-term pairs is defined to be *constant* and *independent* of the tag-term pair's actual content score $s_i(d)$. So a result candidate merely accumulates some additional static score mass if it does *not* match the negated tag-term pair. Let $N \subseteq \{1,\ldots,m\}$ be the set of content conditions marked by a '$-$', then the aggregated score of a candidate item e is defined as

$$score(e,t_i) = \begin{cases} s_i(e) & for\ i \notin N \wedge ftf(t_i,e) > 0 \\ \beta_i & for\ i \in N \wedge ftf(t_i,e) = 0 \\ 0 & otherwise \end{cases} \tag{6}$$

with $\beta_i = 1$ for $i \in N$ and 0 otherwise.

4.5.3 Phrases

Phrases in content conditions are considered as hard conditions, i.e., an element is only considered a match for a content condition with a phrase if it contains the phrase at least once in its full content. Its score is then, for simplicity, the sum of the scores of the phrase's terms. Similarly to the single-term negations, phrase negations are defined to yield a static score mass β_i for each candidate that does not contain the negated phrase. Single-term occurrences of

the negated phrase's terms are allowed, though, and do not contribute to the final element score unless they are also contained in the remaining query.

5 Query Processing

The TopX query processor is responsible for the index-based top-*k* query processing and candidate bookkeeping. The algorithmic skeleton is based on the Combined Algorithm (CA for short, see Section 2.2), which combines sequential scans of inverted index lists with random lookups of index entries. CA uses a round-robin-like – but multi-threaded and batched – *sorted access (SA)* procedure as a baseline. These SA's access the precomputed *inverted lists*, where each inverted list captures all elements satisfying an elementary tag-term condition, sorted by descending score (hence the name "sorted access").

The TopX core algorithm is extended by a *random access (RA)* scheduler to resolve pending conditions by random accesses to specific entries of the inverted lists and other index structures. This enables TopX to resolve navigational and more complex full-text query predicates like phrase conditions that could not (or only with very high costs) be resolved through sorted access to the inverted lists alone.

In this section we focus on the SA procedure for query processing and the score bookkeeping. We postpone a detailed discussion of the RA scheduling to Section 6. In the following subsections, we first introduce the index structures and access primitives that we build on in Subsection 5.1, then present our basic top-*k* query processor and its score bookkeeping for handling text and tag-term content conditions in Subsection 5.2, and finally discuss how to integrate XPath structural conditions by incremental path evaluation in Subsection 5.3.1.

5.1 Index Structures

TopX uses two major kinds of indexes for content conditions and for structural conditions, and also employs a position index for testing textual phrase conditions. All indexes are implemented using a relational DBMS as a storage backend and leveraging its built-in B^+-trees (see Section 10).

The indexes have the following conceptual organization:

– *Tag-term index:* For each tag-term pair we have an inverted list with index entries of the form
  (tag, term, docid, pre, post, level, score, maxscore)
  where *pre* and *post* are the pre/postorder encoding [47] of the element (*pre* is also used as a unique element id within a document), *level* is its depth in the tree, score is the element's score for the tag-term condition, and maxscore is the largest score of any element within the given document for the same tag-term condition. The entries in an inverted list for a (tag,term) pair are sorted in a sophisticated order to aid the query processor, namely, in descending order of the (maxscore, docid, score) values (i.e., using maxscore as a primary sort criterion, docid as a secondary criterion, and score as a tertiary criterion).
– *Structure index:* We encode the locations of elements in documents in a way that gives us efficient tests for the various XPath axes, e.g., to test whether an element is a descendant of another element. To this end we precompute for each tag index entries of the form
  (tag, docid, pre, post, level)
  where *pre* and *post* encode an element's id and navigational position and *level* is the element's depth in its corresponding document tree. These index entries are accessed only by random lookups for given elements.
– *Position index:* For each term we have index entries of the form
  (term, docid, pos)
  where *pos* is the position of the term occurrence in the document. This index is used only for random lookups of such positions in order to test for phrase matches.

The reason for the sophisticated sort order of index entries in the tag-term inverted lists is the following. Our goal is to process matching elements in descending order of scores, according to the TA paradigm, but in addition we would also like to process all elements within the same document and the same tag-term match in one batch as this simplifies the testing of structural conditions (to be discussed below). Ordering the index entries by *(maxscore, docid, score)* yields the highest-scoring matches first but groups all elements of the same document together. Thus, when fetching the next element in score order, we can automatically prefetch all other elements from the same document (within the same inverted list). We refer to this extension of the traditional one-entry-at-a-time scanning as *sorted block-scans*.

An XML element is identified by the combination of the document identifier *docid* and the element's preorder label *pre*. Navigation along the various XPath axes is supported by the *pre* and *post* attributes of the structure-index entries using the technique by [47]. *pre* is the rank of an element in a preorder traversal of the corresponding document tree, and *post* is the rank in postorder traversal. This gives us an efficient test as to whether an element e_1 is an ancestor of another element e_2 (within the same document) by evaluating $pre(e_1) < pre(e_2)$ and $post(e_1) > post(e_2)$, with analogous support for *all* the 13 XPath axes, including the child axis by extending this schema with the *level* information.

Among the above indexes, the tag-term index is used for both sequential scans and random access, whereas the structure index and the position index are used for random access only. When we scan index entries of the tag-term inverted lists, we immediately see not only the id of an element but

actually its full $(pre, post, level)$ coordinates. We can keep this in the candidate cache in memory, and when we later want to compare another encountered element to a previously seen one, we can perform all XPath axis tests in an extremely efficient way. In addition, the structure-index B$^+$-tree provides us with random lookups when needed.

To support also plain text indexing of entire documents (not necessarily only in XML format), we introduce a special virtual element for each document with the reserved virtual tag name $*$, and post a corresponding index entry to the tag-term index. This gives us efficient support for document-level term-only search. Similarly, tag-only lookup is supported by the structure-index access path.

5.2 Basic Top-k Query Processing

In order to find the top-k matches for a CAS query with m content and n structural constraints, scoring, and ranking them, TopX scans all tag-term index lists for the content conditions of the decomposed query in an interleaved manner. Without loss of generality, we assume that these are the index lists numbered L_1 through L_m. In each scan step, the engine reads a large step of b consecutive index entries (with the tunable parameter b typically being in the order of hundreds or thousands). These batches include one or more element blocks that correspond to all elements of the same document in the same index list. The element blocks are then hash-joined with the partial results for the same document previously seen in other index lists. These hash joins take place in memory and immediately test the navigational constraints specified in the query using the pre/post encodings stored in the inverted lists. Also, scores are aggregated and incrementally updated into a *global score* at this point. Note that the way we focus on inexpensive sequential scans leaves *uncertainty* about the final scores of candidates and therefore implies some form of internal bookkeeping and priority queue management not only for the intermediate top-k results, but for all candidates that may still qualify for the final top-k.

When scanning the m index lists, the query processor collects candidates for the query result and maintains them in two such priority queues: one for the *current top-k items*, and another one for *all remaining candidates* that could still make it into the final top-k. The core query processor maintains the following state information:

- the current cursor position pos_i for each list L_i,
- the score values $high_i$ at the current cursor positions, which serve as upper bounds for the unknown scores in the lists' tails,
- a set of current top-k items, d_1 through d_k (renumbered to reflect their current ranks) and a set of documents d_j for $j = k+1..k+q$ in the current candidate queue Q, following a basic data structure containing
  - a set of evaluated query dimensions (i.e., tag-term index lists) $E(d)$ in which d has already been seen during the sequential scans or by random lookups,
  - a set of remainder query dimensions $\bar{E}(d)$ for which the score of d is still unknown,
  - a lower bound $worstscore(d)$ for the total score of d,
  - an upper bound $bestscore(d)$ for the total score of d, which is equal to
$$bestscore(d) := worstscore(d) + \sum_{v \in \bar{E}(d)} high_v \quad (7)$$
    (which is not actually stored but rather computed from $worstscore(d)$ and the current $high_v$ values whenever needed).

Unlike the text-only case considered in the initial TA family, we cannot derive the $worstscore(d)$ bound for a document d simply from the scores for the already evaluated conditions, as this would only be a loose lower bound for the actual value of $worstscore(d)$. Instead, to compute $worstscore(d)$, we have to take into account the structural conditions which makes computing this bound more complex. Section 5.3 explains in detail how worstscore bounds are computed in this case.

In addition, the following information is derived at each step:

- the minimum *worstscore* of the current top-k results, coined *min-k*, which serves as the stopping threshold,
- and for each candidate, a score deficit $\delta(d) = min\text{-}k - worstscore(d)$ that d would have to reach in order to qualify for the current top-k.

The invariant that separates the top-k list from the remaining candidates is that the rank-k worstscore of the top-k queue is at least as high as the best worstscore in the candidate queue. The algorithm can safely terminate, thus yielding the correct top-k results, when the maximum bestscore of the candidate queue is not larger than the rank-k worstscore of the current top-k, i.e., when

$$\underbrace{\min_{d \in top\text{-}k} \{worstscore(d)\}}_{=:\, min\text{-}k} \geq \max_{d \notin top\text{-}k} \{bestscore(d)\} \quad (8)$$

We will refer to Equation 8 as the *min-k threshold test*. More generally, whenever a candidate in the queue Q has a *bestscore* that is not higher than *min-k*, this candidate can be pruned from the queue. Early termination (i.e., the point when the queue becomes empty) is one goal of efficient top-k processing, but early pruning to keep the queue and its memory consumption small is an equally important goal (and is not necessarily implied by early termination). Figure 10 illustrates the corresponding bookkeeping for the intermediate top-k result queue and the candidate queue.

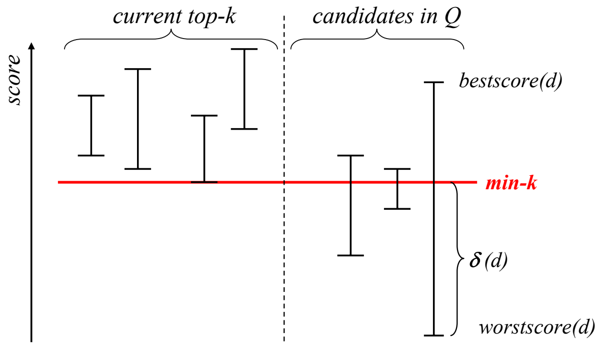

Fig. 10 Top-k and candidate bookkeeping.

We maintain two priority queues in memory to implement the threshold test: one for the current top-k results with items prioritized in ascending order of worstscores, and one for the currently best candidates with items prioritized in descending order of bestscores. The first queue contains only items whose $worstscore(d) \geq min\text{-}k$ and the latter has items whose $worstscore(d) \leq min\text{-}k$ but whose $bestscore(d) > min\text{-}k$.

Note that keeping a large candidate priority queue in memory at any time of the query processing may be expensive. Efficient implementations of the basic TA algorithm may deviate from the strict notion of instantaneously maintained priority queues. Alternative approaches may use a *bounded queue* or merely keep all valid candidates in an *unsorted pool* and iterate over that pool periodically, e.g., after large batches of b sorted access steps (with b in the order of hundreds or thousands of individual index entries) or whenever needed to test the stopping condition.

5.3 Incremental XPath Evaluation

Whenever the index scans have fetched a document's element block for an elementary content condition, we compare this set against other element blocks from the same document, namely those that we have already found through sorted block-scans on index lists for other query conditions. At this point, we compare element blocks for the same document against each other, thus testing structure conditions and aggregating local scores. This is performed efficiently using in-memory hash joins on the *pre* and *post* labels. Documents that have at least one element in each element block satisfying all structure conditions that can be tested so far are kept for later testing of additional conditions; all other candidates can be pruned to save valuable main memory. The document's worstscore is defined as the highest worstscore among the elements (and their embeddings in the document) that match the query's target element.

In *document mode*, we use the worstscore of the rank-k document of the current top-k document list to determine the *min-k* threshold as before; in *element mode*, we use the worstscore of the rank-k element among the current top-k documents to determine the min-k threshold.

5.3.1 Incremental Path Algorithm

The in-memory structural joins for a candidate d are performed *incrementally* after each sequential block-scan on d on a different tag-term index list, i.e., whenever we gain additional information about a candidate document's element structure. In the following we describe the algorithm for these joins in more detail. We introduce a novel approach for incremental path testing that combines hash joins for content-related query conditions and staircase joins (from the XPath Accelerator work [47,48]) for the structure.

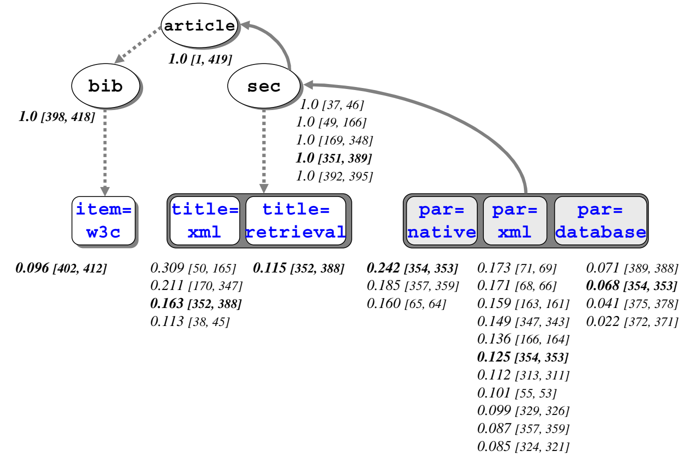

Fig. 11 Path evaluation on a candidate's element structure for the query of Figure 2.

Figure 11 first illustrates the algorithm for a fully evaluated candidate document d and the example query of Figure 2; we will later extend the approach for partially evaluated candidates. The figure shows all element blocks for d depicted as $(score, [pre, post])$ triples for all element blocks that have been mapped to the individual nodes of the query DAG. This is the case when all query conditions have been successfully tested on d by a combination of sorted and random accesses to our indexes. We denote the element block of document d associated with query node n as $elements(d,n)$. For example, the rightmost element block in Figure 11

<u>`par=database`</u>

0.071 [389, 388]
0.068 [354, 353]
0.041 [375, 378]
0.022 [372, 371]

refers to the element block of candidate d for the target content condition `par=database` (including stemming) of the example query of Figure 2. Each of the entries represents a distinct element of the candidate document d. The boldface element entry *0.068 [354, 353]* refers to the `par` element with the *pre* label *354* matching the (stemmed) term "database" among its full-contents. This element has the highest total content score for the three keywords "native", "xml", and "database", and in fact, this element also aggregates the highest overall score with regard to the whole query. Thus, it will determine the document's final score.

The incremental path algorithm performs a recursive tree traversal along the node structure of the query, with individual elements $e \in elements(d,n)$ being joined at each query node n for score aggregation. Note that, although d has valid matches for each of the query conditions, it is still possible that the entire query is not satisfied by d in a conjunctive sense, since at least one element embedding has to form a connected path structure that matches the whole query pattern. Since only those elements that are specified as target elements by the query are defined to be valid result elements and to obtain a non-zero score, we have to *start evaluating* the candidate at the elements matching a *target query condition* which corresponds to the `par` node in our example. Starting with these targets, we traverse the query tree in two opposite directions to make sure we start with a valid result element. For each of the target elements, we aim at *maximizing* the aggregated score of a connected path from a target leaf, via its parent nodes, and down to its valid siblings. The top-scoring target element finally yields the document's score.

In the example structure of Figure 11, we initialize the algorithm by first hash-joining (on the element id) all element blocks for the three query conditions that refer to the target *par* element, namely the `par=native`, `par=xml` and `par=databas` content conditions, grouping all scores for the same element. We see that the `par=native` condition has only few matches with 3 matching elements for that candidate; `par=xml` has the highest number of results, namely, 11 matches; and `par=database` has 4 matches. After hash-joining all three element blocks for that query target dimension, there are still 15 distinct target elements left, each of which is already a valid match for the query (namely, the 15 distinct elements in the union of the three blocks for the `par` element).

For each of these, we have to start a recursive tree traversal for the remaining query dimensions and scores to combine them with elements found for them, using staircase joins to test the structural conditions. For simplicity, we only consider the element with the preorder label 354 (emphasized in boldface) here, which yields the best aggregated score of 0.435 so far. The parent query condition `sec` yields 5 more elements out of which only the one with the preorder label 351 qualifies for further traversal by its pre- and postorder labels. Navigating down from there, we find a `title` element that satisfies the descendant constraint and hence adds a content score of 0.278 to the overall score of element 354. Coming back to the `sec` element, element 354 accumulates a structural score of 1.0 as all structural constraints for the `section` element are satisfied. Similarly, the second parent iteration yields the only `article` root element with a preorder label of 1 and a static local score of 1.0. From here, we recursively navigate down two levels via the `bib` and `item=w3c` query conditions which are also found to provide valid element matches that contribute to the aggregated score of element 354 with values of 1.0 and 0.096, respectively, after checking their pre- and postorder labels. Finally, element 354 obtains an aggregated score of 3.809 which also makes it the top-scored element out of the 15 distinct target elements for the target `par` condition. Note that it is also the only element that satisfies this query in a conjunctive sense.

Conjunctive Mode: The algorithm has the option to terminate an element's evaluation in conjunctive mode if any subtree recursion or single query dimension yields a local score of 0 for the path traversal on that candidate. The evaluation in conjunctive mode considers if at least one query condition i

1) has not been fully evaluated yet ($\exists i \, with \, i \notin E(d)$) through the sequential block-scans, so we do not yet know if the candidate will still satisfy the query conjunctively, and the candidate is kept in the queue,
2) has been tested (i.e., $i \in E(d)$), e.g., through a random lookup, but the inverted list L_i does not contain any match for that document, and the candidate is dropped, or
3) has been tested (i.e., $i \in E(d)$), but there is no valid path from a target element to any of the elements at dimension i based on their pre/postorder labels, and the candidate is dropped.

In all three cases, the document and, thus, all its target elements obtain a $worstscore(d)$ of 0. In the first case, $bestscore(d)$ is assigned a positive value as the document is not yet evaluated at all query dimensions and may still provide a valid path match for all query conditions. Note that we may already take partial knowledge about the candidate's structure into account in order to provide a $bestscore(d)$ bound that is as tight as possible. In the latter two cases, the document obtains also a $bestscore(d) = 0$, and thus the evaluation of d terminates, and d can be safely pruned.

Andish Mode: In andish mode, the evaluation of d is not terminated due to a single failed query condition, but $worstscore(d)$ is increased as soon as one of the query's target elements is positively matched against d and it further increases with more satisfied conditions for further support

elements that are connected to the target element. Similarly, $bestscore(d)$ is not reset to 0 if a single condition fails, but the algorithm assumes that other element blocks for the remaining query conditions may still contribute to the document's score, even if we cannot match any path starting from a target element in the sense of a Boolean XPath-like evaluation anymore.

Unsurprisingly – but in contrast to conventional database queries – conjunctive query evaluations are more expensive to evaluate for a top-k engine than the andish counterpart, because the $[worstscore(d), bestscore(d)]$ intervals converge more slowly and low-scoring content matches cannot be compensated for queries with a drastically reduced conjunctive join selectivity. In the following, we will focus on the andish evaluation strategy as the more interesting but also more difficult case for XML IR involving incremental path validations. The conjunctive mode is kept as an option to support Boolean-XPath-like query evaluations as demanded by some applications.

5.3.2 *Optimizations for Partially Evaluated Candidates*

Now consider the situation with partially evaluated candidates, i.e., with element blocks available in memory for some but not all nodes of the query DAG. To resolve this situation, we introduce the notions of virtual support elements and virtual target elements. They provide means for achieving tighter *worstscore* bounds early in the execution, so that the pruning of candidates works more effectively.

Virtual Support Elements: With only partial knowledge about the document structure, our query processing algorithm could erroneously terminate evaluations when the path structure is interrupted at any node in the query DAG (the same issue would arise with any other XML join algorithm.). In the example structure of Figure 12, the evaluation cannot continue after the hash joins on the target elements as no matches for the `sec` conditions are available; so the *worstscore* bound would be 0.410 (which is way off from the final score of that document, namely, 3.809). Moreover, if the query target node has not yet been evaluated, there would not even be an anchor node to start the evaluation process, because the remainder of the candidate's element structure would simply not be reachable. This would render the *worstscore* and *bestscore* bounds overly conservative and slow down the top-k query processor.

In order to avoid these situations, we introduce the notion of *virtual support elements* for the inner nodes of the query DAG with a local score of 0 and an any-match option for the pre- and postorder-based staircase joins, thus conceptually attaching entries of the form

0.0 [, *]*

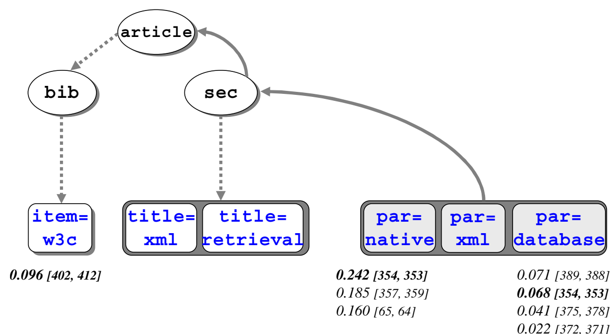

Fig. 12 Partially evaluated candidates.

to each element block. These "wildcard" elements may be joined with any "real" element-block or with other virtual support elements for navigation through unevaluated navigational query conditions. Even after an element-block for an inner node is fetched from disk, we keep the virtual navigational element for that node. This way, the content nodes serve as synapses for connecting subtrees, without having to necessarily make the actual random lookup for the connecting path condition. In andish evaluation mode, we can now safely increase the *worstscore* of a candidate d without having to assume a connected path structure, and this allows us to compute tighter *worstscore* and *bestscore* bounds taking into account *all* the evaluated query conditions. In many cases, the bestscore of a candidate document based on its content-related query conditions might already make it eligible for pruning without having to perform the actual random lookups for the structural conditions.

Virtual Target Elements: As mentioned above, the lack of a target element in a candidate's currently known element structure would prevent our algorithm from further processing the candidate and would keep the *bestscore* bound unnecessarily high. Similarly to the virtual support elements, which mainly serve to reason more accurately about a candidate's worstscore, we also introduce the notion of *virtual target elements* with a local score of 0 and the same any-match option for the pre- and postorder-based staircase joins. The difference between the two kinds of virtual elements is that a virtual target element helps us to more accurately restrict a candidate's *bestscore*, whereas the *worstscore* has to be zero anyway as long as no valid target element has been detected.

6 Random Access Scheduling

The query processing algorithm presented in the previous section is driven by sequential scans on the tag-term index lists, and its incremental path evaluation aims to piggyback

the testing of structural conditions on these scans or postpone them altogether. However, there are various situations in which extra tests with higher costs are unavoidable. We refer to such tests as *expensive predicates*. Probing candidates as to whether they satisfy such a predicate will involve *random accesses (RA's)* to structure indexes and possibly other disk-resident data structures. In this section, we will first characterize expensive predicates in Subsection 6.1, and then we will discuss heuristics for scheduling the necessary probing steps in Subsection 6.2. In addition, it turns out that we can turn the impediment of having to perform some random accesses into an opportunity: sometimes it can be beneficial from an overall efficiency viewpoint to deviate from the sequential scan strategy and schedule judiciously chosen random-access steps at appropriate points even for tag-term conditions. In certain situations, such a mixed strategy of interleaving sequential accesses with random accesses allows pruning of result candidates and may terminate the threshold algorithm much earlier. Therefore, we have developed also a cost-based scheduling method for deciding when to issue random accesses; this will be presented in Subsection 6.3.

6.1 Expensive Predicates

Certain structural conditions in queries cannot be tested by sorted access to the tag-term index alone, and TopX then resorts to random accesses on the structure index. Also, auxiliary query hints in the form of expensive text predicates like phrases ("..."), mandatory terms (+), and negations (−) are often used to improve the retrieval effectiveness and may require random accesses to disk-resident data structures. The challenge for a top-k query processor lies in the *efficient* implementation of these additional query constraints and their integration into the sorted vs. random access scheduling paradigm. Thus, generalizing the notion of expensive predicates defined in [26], a query predicate is considered *expensive* if it cannot be resolved at all through sorted access alone or relying on sorted access alone would entail very high costs (e.g., because it would need scan an index list (almost) to its very end).

It follows that tag-only structural conditions (i.e., without an associated content term) are expensive, because they cannot be tested with inverted indexes on tag-term pairs at all but require random accesses to the structure index. The `author//affiliation` condition in the query `//book[about(., Information Retrieval XML)]//author//affiliation` would be an example. Phrase tests are expensive because they require additional accesses to positional information that is not included in the inverted lists, and negations are expensive because, unless we performed a random access to test the absence of an element in an inverted list, we would have to scan entire lists regardless of the document's score in a negated condition.

6.2 Min-Probe Heuristics

The key idea of our probing heuristics is to postpone the testing of expensive predicates by RA's as much as possible, and perform these tests only when their evaluation would push a candidate into the current top-k results. To this end we maintain a *score gap* value for each candidate, which is the additional score mass that the candidate would immediately earn if we now learned that all expensive predicates were true. To this end we extend our run-time data structures by two additional bit vectors for each candidate in the pool:

- $P(d)$: a set of unevaluated expensive predicates in content conditions, e.g., phrase conditions, that d still has to match, and
- $O(d)$: a set of unevaluated structural conditions that d still has to match.

$P(d)$ and $O(d)$ are dependent on the structure of the query at hand. Suppose the query has m content conditions and n structural conditions. Then initially $P(d) \subseteq \{1,\ldots,m\}$ contains those content conditions that refer to expensive predicates such as phrase conditions, and $O(d)$ is initialized by $\{1,\ldots,n\}$, i.e., contains all structural conditions in the query. For example, in the query

//article//sec[about(.,undersea "fiber optics cable" -satellite)

the initial $P(d)$ for each candidate contains the conditions "sec=fiber" (2), "sec=optics" (3), "sec=cable" (4) because of the phrase condition, and "sec=satellite" (5) for the negation, but not "sec=undersea" (1) as this is not involved in any expensive predicate; thus we set $P(d) = \{2,3,4,5\}$.

We define the score gap $gap_P(d)$ that a candidate d can earn for the conditions in $P(d)$ as the accumulated score from content conditions that have already been evaluated on the tag-term index with scores $s_i(e)$ for elements in d:

$$gap_P(d) = \sum_{i=1}^{m} s_i(e) \quad for\ i \in E(d) \cap P(d) \tag{9}$$

Without knowing that the conditions in $P(d)$ are actually satisfied, the *worstscore* bookkeeping for the candidate d could not consider this score mass. Only when we know that the expensive predicates in $P(d)$ do indeed hold, we can safely increase the *worstscore* of d by $gap_P(d)$.

Analogously, we define the score gap $gap_O(d)$ as the score mass that a candidate would accumulate if all structural conditions in $O(d)$ were now known to be true for the candidate:

$$gap_O(d) = \sum_{i=1}^{n} c \tag{10}$$

where c is the constant defined in Section 4.2. The overall gap $gap(d)$ of a document d is then defined as the sum of $gap_P(d)$ and $gap_O(d)$. The gap of a document represents the

maximal additional score the document would achieve if all expensive predicates were evaluated to true.

In order to keep the updates for a candidate's score bounds monotonic, the lower $worstscore(d)$ bound of a candidate must not include any evaluated conditions that belong to an expensive predicate, i.e., it can only consider conditions in $E(d) \setminus P(d)$. So we are conservative on the worst-score bound. The bestscore bound, on the other hand, remains unaffected, because, even when we have not yet tested a predicate, we can be sure that d will not accumulate *more* score mass than we already assumed for the best possible case.

Now we are in a position to define our *Min-Probe* scheduling heuristics: we schedule the RA's for all $i \in P(d)$ only if

$$worstscore(d) + gap(d) > min\text{-}k \tag{11}$$

which is the natural adaptation of the necessary-predicate-probe strategy of [26] to our setting where SA's are considered inexpensive and RA's expensive.

The value of $gap(d)$ increases whenever we see a candidate in list $i \in E(d)$ during the index scans; so our heuristic scheduling criterion tends to be fulfilled only at a *late* stage of the query processing for most candidates. In fact, we schedule RAs for the unresolved predicates on d only if we know that this will promote the candidate to the (current) top-k results and will then lead to an increase of the $min\text{-}k$ threshold (which in turn would typically lead to an increased pruning of remaining candidates). This way, only the most promising candidates are tested; for the great majority of candidates, $worstscore(d) + gap(d)$ will never exceed $min\text{-}k$.

Note that a sequence of RA's to test multiple expensive predicates for the same candidate can be terminated as soon as $bestscore(d) \leq min\text{-}k$, i.e., the candidate fails on sufficiently many conditions and is then dropped from the queue. This additional optimization is easily implemented using our run-time data structures for bookkeeping.

6.3 Cost-Based Random Access Scheduling

While the *Min-Probe* scheduling heuristics presented in the previous subsection is light-weight in terms of overhead, it does not take into account the actual benefit/cost ratio of random vs. sorted accesses and would never consider RA's for tag-term conditions. This subsection presents the *Ben-Probe* scheduler that applies a cost model to choose the next operation among sorted accesses, random accesses for content conditions, and random accesses for expensive predicates (limited to structural conditions here for clarity of presentation).

Ben-Probe estimates the probability $p(d)$ that document d, which has been seen in the tag-term index lists $E(d)$ and has not yet been encountered in lists $\bar{E}(d) = [1..m] - E(d)$, qualifies for the final top-k result by a combined score predictor and selectivity estimator. These predictors are explained below.

We break down the query structure into the following basic subquery patterns:

- *tag-term pairs*: for content conditions
- *descendants*: tag pairs for transitively expanded descendant conditions
- *twigs*: tag triples of branching path elements for transitively expanded descendant conditions

We estimate, whenever we consider scheduling RA's for a candidate d, the selectivity of the $o(d)$ not yet evaluated navigational conditions using these patterns.Here the selectivity σ_i of a navigational condition o_i is the estimated probability that a randomly drawn candidate satisfies the navigational condition o_i. We estimate these selectivities by precomputed corpus frequencies of ancestor-descendant and branching path elements, i.e., pairs and triples of tags. Note that this is a simple form of an XML synopsis for this kind of statistics management. It could be replaced by more advanced approaches such as those in [1,65,76,101], but our experiments indicate that our simple approach already yields a very effective method for pruning and identifying *which* candidate should be tested *when* by explicitly scheduled RA's.

The Ben-Probe scheduler compares the cost of making random accesses 1) to inverted tag-term index lists or 2) to indexes for navigational conditions versus 3) the cost of proceeding with the sorted-access index scans. For all three cost categories, we consider 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 (hypothesized) optimal schedule that could make random lookups only for the final top-k and would traverse index lists with different and minimal depths.

For looking up unknown scores of a candidate d in the index lists $\bar{E}(d)$, we would incur $|\bar{E}(d)|$ random accesses which are wasted if d does not qualify for the final top-k result (even after considering the additional score mass from $E(d)$). We can estimate this probability as

$$\begin{aligned} P[d \notin \ top\text{-}k] &= 1 - p(d) \\ &= 1 - p_S(d) \cdot q(d) \ , \end{aligned} \tag{12}$$

where $q(d)$ is our selectivity estimator (see below) and $p_S(d)$ is the score predictor

$$p_S(d) = P\left[\sum_{i \in \bar{E}(d)} S_i > \delta(d) \mid S_i \leq high_i\right] \tag{13}$$

where $\delta(d) = min\text{-}k - worstscore(d) - o(d) \cdot c$, S_i denotes the random variable which captures the probabilistic event that document d has a score of $s_i(d)$ for content condition i, and $o(d)$ is the number of currently unevaluated navigational

conditions for d. Since this may involve the sum of two or more random variables, this entails computing the *convolution* of the corresponding index lists' score distributions to compute this probability, using either a parameterized score estimator or compact and flexible histograms (see [93] for details). $q(d)$ is a correlation-aware selectivity estimator

$$q(d) = \left(1 - \prod_{i \in \bar{E}(d)} \left(1 - \max_{j \in E(d)} \frac{l_{ij}}{l_j}\right)\right) \tag{14}$$

where l_j denotes the length of list L_j, and l_{ij} denotes the (estimated) number of documents that occur in both L_i and L_j (see [18] for a more detailed derivation). Note that this way, we are able to incorporate any available information about score convolutions, index list selectivities, and correlations between tag-term pairs into the final estimation of $p(d)$.Then the random accesses to resolve the missing tag-term scores have expected wasted cost:

$$EWC_{RA\text{-}C}(d) := |\overline{E}(d)| \cdot (1 - p_S(d) \cdot q(d)) \cdot \frac{c_R}{c_S} \tag{15}$$

where $\frac{c_R}{c_S}$ is the cost ratio of RA's and SA's.

As for path conditions, the random accesses to resolve all $o(d)$ navigational 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. Recall from our scoring model that each satisfied navigational condition earns a static score mass c. Denoting the set of unevaluated navigational conditions as Y, we can compute the probability $q'(d)$ that a candidate d accumulates enough score mass for navigational constraints to achieve, together with additional scores from content conditions, a score above min-k:

$$q'(d) = \sum_{Y' \subseteq Y} P[Y' \; is \; satisfied] \cdot P\left[\sum_{i \in \overline{E}(d)} Si > min\text{-}k - worstscore(d) - |Y'| \cdot c\right]$$

where the sum ranges over all subsets Y' of the remaining navigational conditions Y. $P[Y' \; is \; satisfied]$ is estimated as

$$P[Y' \; is \; satisfied] = \prod_{v \in Y'} \sigma_v \cdot \prod_{v \notin Y'} (1 - \sigma_v) \;, \tag{16}$$

assuming independence for tractability; here, the σ_v are the selectivities of the unevaluated navigational conditions. For efficiency, rather than summing up over the full amount of subsets $Y' \subseteq Y$, a lower-bound approximation can be used. That is, we do not consider all subsets Y' but only those that correspond to a greedy order of evaluating the navigational conditions in ascending order of selectivity, thus yielding a lower bound for the true cost. Then the random accesses for path and twig conditions have expected wasted cost:

$$EWC_{RA\text{-}S}(d) := o(d) \cdot q'(d) \cdot \frac{c_R}{c_S} \tag{17}$$

The next batch of b sorted accesses to each content-related index list incurs a fractional cost for each candidate in the priority queue, and the total cost is shared by all candidates in the candidate priority queue Q. For a candidate d, the sorted accesses are wasted if either 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 lists in $\bar{E}(d)$, or if we encounter d, but it does not make it to the top-k. The probability $q_i^b(d)$ of *not* seeing d in the ith list in the next b steps is defined as

$$\begin{aligned} q_i^b(d) &= 1 - P[d \text{ in next } b \text{ elements of } L_i \mid i \in E(d)] \\ &= 1 - \frac{l_i - pos_i}{n} \cdot \frac{b}{l_i - pos_i} \\ &= 1 - \frac{b}{n} \end{aligned} \tag{18}$$

where l_i is the length of the ith list, pos_i is the current scan position in that list, and n is the number of documents.
We can compute the probability $q^b(d)$ of seeing d in *at least one* list in the batch of size b as:

$$\begin{aligned} q^b(d) &= 1 - P[d \text{ not seen in any list}] \\ &= 1 - \prod_{i \in \bar{E}(d)} q_i^b(d) \end{aligned} \tag{19}$$

So the probability of not seeing d in any list is $1 - q^b(d)$. The probability that d is seen in at least one list, but does not make it into the top-k, can be computed as

$$q_S^b(d) := (1 - p_S(d)) \cdot q^b(d) \tag{20}$$

The total costs for the next batch of b sorted accesses in each of the m tag-term index lists is shared by all candidates in Q, and this finally incurs expected wasted cost:

$$\begin{aligned} EWC_{SA} &:= \frac{b \cdot m}{|Q|} \cdot \sum_{d \in Q} q_S^b(d) \\ &= \frac{b \cdot m}{|Q|} \cdot \sum_{d \in Q} \Big((1 - q^b(d)) + (1 - p_S(d)) \cdot q^b(d)\Big) \\ &= \frac{b \cdot m}{|Q|} \cdot \sum_{d \in Q} \Big(1 - p_S(d) \cdot q^b(d)\Big) \end{aligned} \tag{21}$$

We initiate the random accesses for tag-term score lookups and for navigational conditions for a candidate d if and only if

$$EWC_{RA\text{-}C}(d) < EWC_{SA} \;\wedge\; EWC_{RA\text{-}S}(d) < EWC_{SA}$$

with RA's weighted to SA's according to the cost ratio c_R/c_S. We actually perform the random accesses one at a time in ascending order of content-related (for tag-term pairs) and structural selectivities (for navigational conditions). Candidates that can no longer qualify for the top-k are eliminated as early as possible and further random accesses for them are canceled.

7 Probabilistic Candidate Pruning

The TA-style *min-k* threshold test is often unnecessarily conservative, because the *expected* remainder score of a document is typically much lower than the actual sum of the $high_i$ bounds for $i \notin E(d)$ at the current scan positions, which could make more candidates eligible for pruning at an early stage of the query processing. Of course, using plain expectations for pruning would not give us guarantees for not missing any of the true top-*k* results. But we would expect that the final sum of the $s_i(d)$ scores in the remainder set $\bar{E}(d)$ is lower than the sum of the $high_i$ bounds with very high probability. Thus, we refer to Equation 22 as the *probabilistic threshold test*:

$$p(d) \leq \varepsilon \tag{22}$$

That is, if the probability $p(d)$ (the probability that the document qualifies for the final top-*k* introduced in the previous section) was below some threshold ε, e.g., between 1 and 10 percent, then we might decide to disregard d and drop the candidate from the queue without computing its full score, thus introducing a notion of approximate top-*k* query processing but with a controlled pruning aggressiveness for which we can derive *probabilistic guarantees* for the result precision.

The previous considerations provide us with score predictions for individual candidate items at arbitrary steps during the sequential index accesses. These probabilistic predictions in our query processing strategies lead to probabilistic guarantees from a user viewpoint, if we restrict the action upon a failed threshold test to dropping candidates, but we still stop the entire algorithm only if the entire queue runs out of candidates. In this case the probability of missing an object that should be in the true top-*k* result is the same as erroneously dropping a candidate, i.e., pruning errors are assumed to be uniformly distributed among all items discovered during index processing; and this error, call it p_{miss}, is bounded by the probability ε that we use in the probabilistic predictor when assessing a candidate. For the *relative* recall of the top-*k* result, i.e., the fraction of true top-*k* objects that the approximate method returns, this means that

$$\begin{aligned} &P[\, recall = r/k \,] = \\ &P[\, precision = r/k \,] = \\ &\qquad = \binom{k}{r} (1 - p_{miss})^r \, p_{miss}{}^{(k-r)} \\ &\qquad \leq \binom{k}{r} (1-\varepsilon)^r \varepsilon^{(k-r)} \end{aligned} \tag{23}$$

where r denotes the number of correct results in the approximate top-*k*. We can then efficiently compute Chernoff-Hoeffding bounds for this binomial distribution.

Note that the very same probabilistic guarantee holds for the precision of the returned top-*k* result, simply because recall and precision use the same denominator k in this case. The predicted expected precision then is

$$\begin{aligned} E\,[precision] &= \sum_{r=0}^{k} P\,[\, precision = r/k \,] \cdot \frac{r}{k} \\ &= 1 - \varepsilon \end{aligned} \tag{24}$$

This result yields a compact and intuitive assumption on the result quality that the approximate top-*k* algorithms provides compared to the exact top-*k* algorithm without probabilistic pruning in terms of *relative* precision or recall, i.e., the overlap of two result sets.

In practice, the score differences between the top-ranked items are often very marginal for many real-world, large corpora and scoring models such as TF·IDF or BM25. Our experiments on various data collections using human relevance judgments for query results indicate that with increasing pruning aggressiveness, the user-perceived result quality decreases at a much lower rate than the relative overlap measures.

8 Dynamic & Incremental Query Expansion

Query expansion is a successful method to improve recall for difficult queries. Traditional query expansion methods select expansion terms whose thematic similarity to the original query terms are above some specified threshold, e.g., using the Rocchio [80] method or Robertson and Spärck-Jones [78] weights, thus generating a non-conjunctive (or "andish") query of much higher dimensionality. However, these methods typically incur three disadvantages: (1) the threshold for selecting expansion terms needs to be carefully handtuned for each query, (2) an inappropriate choice of the threshold may result in either not improving recall (if the threshold is set too conservatively) or in topic dilution (if the query is expanded too aggressively), and (3) the expansion may often result in queries with a large number of terms, which in turn leads to poor efficiency when evaluating such expanded queries. For XML, these problems are even worse, as not only terms can be expanded, but also tags; while we focus on term expansions in this paper, the proposed techniques can be applied for expanding tags as well.

The query expansion approach used in TopX addresses all three problems by *dynamically* and *incrementally merging* the inverted lists for the potential expansion terms with the lists for the original query terms. We introduce a novel notion of *best match* score aggregation that only allows for the best match per expansion group to contribute to the final score, thus reflecting the semantic structure of the query directly in the query processing and score aggregation. The algorithm is implemented as an Incremental Merge operator that can be smoothly integrated with the query processing

framework presented before. In the following, we first introduce thesaurus-based query expansion in Subsection 8.1. Subsection 8.2 then shows how incremental expansion of single query terms is integrated in the query processing, and Subsection 8.3 explains how expansions of phrases are processed.

8.1 Thesaurus-based Query Expansion

We generate potential expansion terms for queries using a thesaurus database based on WordNet [42]. WordNet is the largest electronically available, common-sense thesaurus with more than 120,000 semantic concepts, consisting of single terms and as well as explicitly identified phrases, and more than 300,000 handcrafted links that define the way how the concepts or *synsets* (i.e., sets of synonyms that refer to the same meaning) in the WordNet graph are related. The basic structure of WordNet with regard to the hypernym relationship is essentially that of a tree which is the reason why WordNet is often referred to as a hierarchical thesaurus (HT).

8.1.1 Word Sense Disambiguation

Query expansion techniques used in IR typically suffer from the following two common phenomena of word usage in natural language:

1) *Polysemy:* A term can have different meanings depending on the context that it is used in.
2) *Synonymy:* Multiple terms have the same meaning; together with 1) the situation may become mutually context sensitive.

In order to address these problems, a query term t is mapped onto a WordNet concept c by comparing some form of textual context of the query term (i.e., the description of the query topic or the summaries of the top-10 results of the original query when relevance feedback is available) against the context of synsets and glosses (i.e., short descriptions) of possible matches for c and its neighbors in the ontology graph. The mapping uses a simple form of *Word Sense Disambiguation* (WSD) by choosing the concept with the highest similarity of each two context pairs.

As an example for our efforts, consider the term "goal" which yields the following different word senses when queried in WordNet:

1) $\{$*goal, end,...*$\}$ – the state of affairs that a plan is intended to achieve and that (when achieved) terminates behavior to achieve it; "the ends justify the means"
2) $\{$*goal*$\}$ – a successful attempt at scoring; "the winning goal came with less than a minute left to play"

and two further senses. By looking up the synonyms of these word senses, we can construct the synsets $\{$*goal, end, content, cognitive content, mental object*$\}$ and $\{$*goal, score*$\}$ for the first and second meaning, respectively. As each of the meanings is connected to different concepts in the ontology graph, a reliable disambiguation and choice of the seed concepts is a crucial precondition for any subsequent expansion or classification technique.

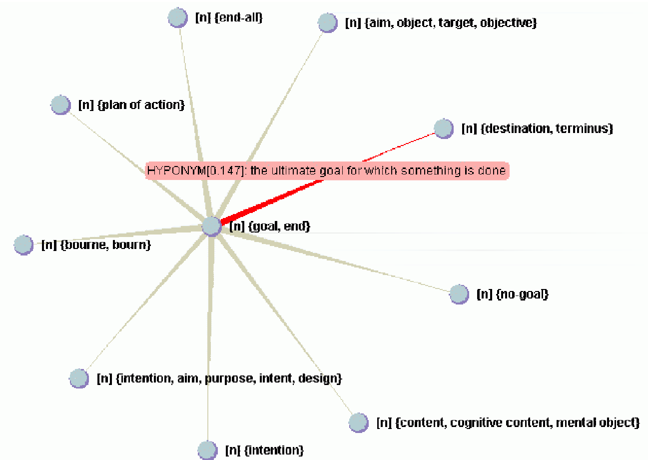

Fig. 13 Visualization of the concept neighborhood graph for one possible meaning of the word 'goal'.

Now the key question is of course: which of the possible senses of a word is the right one? Our approach to answer this question is based on word statistics for some local context of both the term that appears in a part of a document or a keyword query and the candidate senses that are extracted from the concept graph.

8.1.2 Independent Mapping

In [89], we coined the first approach the *Independent Mapping* or *Independent Disambiguation*, because each term or n-gram (i.e., a set of n adjacent terms) out of a given word sequence (which is a keyword query in our application) is mapped individually onto its most likely meaning in the concept graph without taking the mapping of the sequence "as a whole" into account (considering also the relationships between the mappings of these terms or n-grams).

In order to identify the largest possible subsequences of n-grams out of a given sequence, let us first consider a word sequence $w_1, ..., w_m$. Starting with the first word w_1 at position $i = 1$ in the sequence and a small lookahead distance m' of at most 5 words, we use a simple window parsing technique to determine the largest subsequence of words that can be matched with a phrase contained in WordNet's synsets to identify an initial set of possible word senses $s_{i_1}, \ldots, s_{i_p}$. If we have successfully matched the current sequence

$w_i, \ldots, w_{i+m'}$, we increment i by m' and continue the mapping procedure on the suffix of the sequence; if we could not match the current sequence onto any phrase denoted by a WordNet concept, we decrement m' by 1 and try the lookup again until $m' = 1$. After performing that subroutine, i is again incremented by 1 until $i = m$. Fortunately, phrases of length 2 or 3 hardly ever exhibit more than one distinct meaning in WordNet, whereas in fact most single keywords match more than one semantic concept and, thus, are highly ambiguous.

For a given term or n-gram t, we consider the query that it occurred in as the *local context* $con(t)$ *of* t. For a candidate word sense s, we extract synonyms, all immediate hyponyms and hypernyms, and also the hyponyms of the hypernyms (i.e., the siblings of s in the HT). Each of these has a synset and also a short explanatory text, coined "gloss" in the WordNet terminology. We form the union of the synsets and corresponding glosses, thus constructing a *local context* $con(s)$ *of sense s* extracting also n-grams from synsets and glosses. As an example, the context of sense 1 of the word "goal" (see Figure 13) corresponds to the bag of words {*goal, end, state, affairs, plan, intend, achieve, ..., content, cognitive content, mental object, perceived, discovered, learned, ..., aim, object, objective, target, goal, intended, attained, ...* }, whereas sense 2 would be expanded into {*goal, successful, attempt, scoring, winning, goal, minute, play, ..., score, act, game, sport, ...* }.

The final step toward disambiguating the mapping of a term onto a word sense is to compare the term context $con(t)$ with the context of candidate concepts $con(s_1)$ through $con(s_p)$ in terms of a similarity measure between two bags of words. The standard IR measure for this purpose would be the cosine similarity between $con(t)$ and $con(s_j)$, or alternatively the Kullback-Leibler divergence [17] between the two word frequency distributions (note that the context construction may add the same word multiple times, and this information is kept in the word bag). Our implementation uses the Cosine [17] similarity between the TF·IDF vectors of $con(t)$ and $con(s_j)$ for its simpler computation.

Finally, we map term t onto that sense s_j whose context has the highest similarity, i.e., the lowest cosine distance, to $con(t)$. We denote this word sense as $sense(t)$. If there is no overlap at all, e.g., if the context denoted by a keyword query consists only of a single term, namely the one that is about to be expanded, we choose the sense that has the highest a-priory probability, i.e., the one with the lowest IDF-value.

8.1.3 Edge Similarities

There have been various efforts proposed in the literature aiming to quantify semantic similarities of concepts in WordNet [42]. We believe that among the most promising ones are those that aim to model concept similarities on the basis of term and phrase correlations over large, real-world data collections. These measures exploit co-occurrence statistics for terms (or n-gram phrases) to estimate the semantic relatedness of terms and, hence, concepts in a given corpus. Ideally, this is the same corpus that is also used for querying. A measure often referred to for this purpose is the *Dice* coefficient.

As for Dice coefficients, the similarity between to senses S_1 and S_2 is defined as:

$$dice(S_1,S_2) := max_{S_1 \times S_2} \left\{ 2 \cdot \frac{df(t_{1,i} \wedge t_{2,j})}{df(t_{1,i}) + df(t_{2,j})} \right\} \quad (25)$$

where $t_{1,i} \in S_1$ and $t_{2,j} \in S_2$, respectively, and $df(t_{1,i} \wedge t_{2,j})$ is the cardinality of documents that contain both $t_{1,i}$ and $t_{2,j}$.

8.1.4 Path Similarities

To implement the similarity search for two arbitrary, not directly connected concepts S_1 and S_2, we employ Dijkstra's shortest path algorithm [33] to find the shortest connection between S_1 and S_2. Then, interpreting the edge similarities as transition probabilities, the senses' final path similarity $sim(S_1,S_2)$ for a path $\langle v_1, \ldots, v_k \rangle$ of length k with $v_0 = S_1$ and $v_k = S_2$ and $\langle v_i, v_{i+1} \rangle \in V$ for $i = i, \ldots, k-1$ is defined as

$$sim(S_1,S_2) := \prod_{i=1}^{k-1} dice(v_i, v_{i+1}) \quad (26)$$

If there is more than one path that minimizes the length, we choose the one with highest path similarity *sim* to yield the final concept similarity.

8.2 Incremental Merge Operator

TopX can either automatically expand all terms and/or tags in a query or only those where the user requested expansion; this is done using the $\sim$ operator as in the query `//article[about(~title,~xml)]`. For simplicity, we discuss only expansion of terms; the expansion of tags can be implemented analogously.

For an elementary content condition of the form $A = \sim t_i$ and an expansion set $exp(t_i) = \{t_{i1}, \ldots, t_{ip}\}$ with corresponding similarities $sim(t_i, t_{ij})$, we merge the corresponding p inverted index lists $L_{i1} \ldots L_{ip}$ in descending order of the combined maxscore that results from the maximum local score $s_{ij}(d)$ of an expansion term t_{ij} in any element of a document d and the thesaurus-based similarity $sim(t_i, t_{ij})$, keeping the block structure intact. Moreover, to reduce the danger of topic drift, we consider for any element e only its maximum combined score from any of these lists, i.e.,

$$score(e, A = \sim t_i) := \max_{t_{ij} \in exp(t_i)} sim(t_i, t_{ij}) \cdot score(e, A = t_{ij}) \quad (27)$$

with analogous formulations for the $worstscore(d)$ and $bestscore(d)$ bounds as used in the baseline top-k algorithm.

The actual set of expansions is typically chosen such that for a content condition A=$\sim t_i$, we first look up the potential expansion terms $t_{ij} \in exp(t_i)$ with $sim(t_i, t_{ij}) > \theta$, where θ is a fine-tuning threshold for limiting $exp(t_i)$. It is important to note that this is not the usual kind of threshold used in query expansion; it is merely needed to upper-bound the potential expansion sets and to yield a baseline for comparisons to a static expansion technique. Then the index lists for the expanded content conditions A=t_i1, A=t_{i2}, ... are merged on demand (and, hence, incrementally) until the min-kthreshold termination at the enclosing top-k operator is reached, by using the following scheduling procedure for the index scan steps: the next scan step is always performed on the list L_{ij} with the currently highest value of $sim(t_i, t_{ij}) \cdot high_{ij}$, where $high_{ij}$ is the last score seen in the index scan (i.e., the upper bound for the unvisited part of the list). This procedure guarantees that index entries are consumed in exactly the right order of descending $sim(t_i, t_{ij}) \cdot s_{ij}(d)$ products. Figure 14 illustrates this process. Here, the scan starts with the list for A=t_1 with a combined upper bound of $1.0 \cdot 0.9 = 0.9$ and continues until the block for document $d1$ is encountered. As the maxscore of this block ($1.0 \cdot 0.4 = 0.4$) is below the upper bound of the list for A=t_2 ($0.9 \cdot 0.8 = 0.82$), the scan continues in that list.

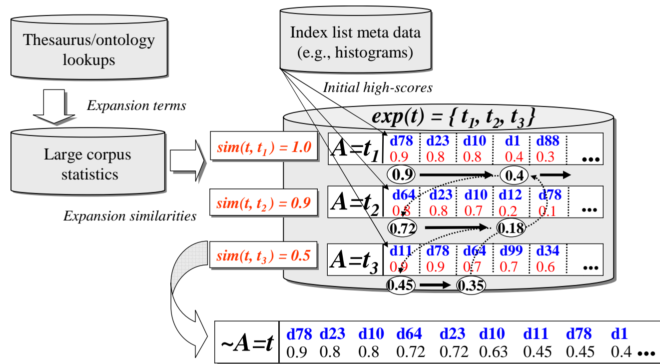

Fig. 14 Example schedule for Incremental Merge.

The scans on the expansionindex lists are opened *as late as possible*, namely, when we actually want to fetch the first index entry from such a list. Thus, resources associated with index-scan cursors are also allocated on demand.

As a side effect of combining multiple lists, documents and elements may occur multiple times in the merged output. We cannot easily drop all but the first occurrence of a document as different elements from following blocks in other lists could satisfy the structural constraints and lead to subtrees with higher total scores. Hence, dropping them would make us run into the danger of reporting false negatives and potentially prune candidates from the top-level queue too early. However, these potential matches can easily be detected through further merging the expanded lists and iteratively polling the Incremental Merge operator for the next element block in descending order of the combined similarity and block scores. This yields a new, more conservative bestscore bound that considers for already evaluated incremental merge dimensions, instead of the already known score, the maximum of this score and the current $high_i$ bound for this dimension.

8.3 Evaluating Expanded Phrases

If a term is expanded into at least one phrase, local scores for this expansion cannot be fetched from materialized index lists but need themselves to be computed dynamically. This poses a major problem to any top-k algorithm that wants to primarily use sorted accesses. A possible remedy would be that the global top-k operator "guesses" a value k' and asks the dynamic source to compute its top-k' results upfront, with k' being sufficiently large so that the global operator never needs any scores of items that are not in the local top-k'. We believe that this is unsatisfactory, since it inherently is very difficult to choose an appropriate (i.e., safe and tight) value for k', and this approach would destroy the incremental and pipelined nature of our desired operator architecture.

TopX treats such situations by running a nested top-k operator on the dynamic data source(s), which iteratively reports candidates to the caller (i.e., the global top-k operator), and efficiently synchronizes the candidate priority queues of caller and callee. The callee starts computing a top-∞ result in an incremental manner, by whatever means it has; in particular, it may use a TA-style method itself without a specified target k, hence top-∞. It gradually builds a candidate queue with $[worstscore'(d), bestscore'(d)]$ intervals for each candidate d. The caller periodically polls the nested top-k operator for its currently best intermediate results with their score intervals. Now the caller integrates this information into its own bookkeeping by adding bestscores to the bestscores of its global candidates and worstscores to the worstscores of its global candidates. From this point, the caller's processing simply follows the standard top-k algorithm (but with score intervals). This method nicely provides a non-blocking pipelining between caller and callees, and gives the callees leeway as to how exactly they proceed for computing their top results. Note that the caller may terminate (and terminate all callees) long before a callee has really computed its final top results.

Within the TopX engine, nested top-k operators are primarily useful for handling phrase matching in combination with query expansion. In general, it will be too expensive to precompute and materialize an inverted list for all possible phrases. But if we merely index the individual words, we

cannot simply look up the combined scores in local index lists as we would need for an Incremental Merge. Our solution is to encapsulate phrase conditions in separate top-k operators and invoke these from the global top-k operator in the pipelined manner described above.

For an Incremental Merge expansion that includes at least one phrase, we incrementally merge lists of partially evaluated candidates obtained from a nested top-k operator for each phrase in descending order of candidate bestscores. For single-keyword expansions, the score obtained from a single inverted list will already be the final score for that candidate and expansion; for phrase expansions, the score will be a partial score obtained from one or more local keyword conditions of the phrase.

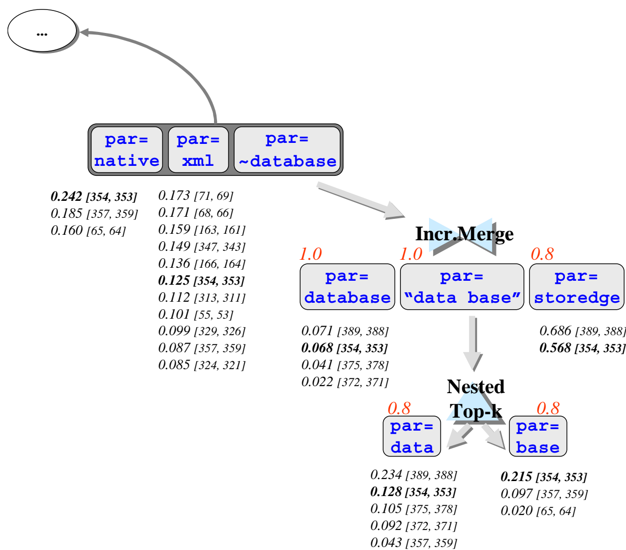

Fig. 15 Dynamic expansion of a content condition with a phrase.

Figure 15 depicts the situation for an example expansion of the tag-term pair `par=database` into `par=database`, `par=‘‘data base’’`, and `par=storedge` having similarities 1.0, 1.0, and 0.8, respectively. A nested top-k operator is utilized to generate a dynamic index list for the phrase expansion `par= ‘‘data base’’` which aggregates phrase scores with respect to individual element scores. Phrase tests are now used to prune individual elements and do not necessarily render the whole candidate document invalid, when the test failed only for some of the elements.

9 Hybrid Index Structures

As shown in Section 5.1, the pre/postorder labeling scheme can efficiently evaluate the descendant axis in location paths. However, it might degenerate for deeply nested path expressions with low selectivity (i.e., few matches), because they require many joins to evaluate. Data-guides [44], on the other hand, with their ability to encode entire location paths into a single label or *bucket id*, are a perfect method to address this issue, but they do not support the descendant axis in location paths well. Although we might try to precompute all descendant path relaxations and materialize them in our inverted index for all bucketid-term pairs, this would hardly be feasible for an XML collection with a complex schema or diverse structure such as the INEX IEEE collection.

As an example, consider the seemingly inconspicuous path expression

```
//article//sec//p
```

which contains three descendant-axis steps and yields exactly 520 distinct bucket ids (i.e., distinct root-to-leaf paths) in the data-guide structure for the INEX IEEE collection. Thus, an intriguing idea would be to perform the relaxation (for a reasonable amount of choices in the expansion possibilities) again directly in the query processor, now using the Incremental Merge approach to dynamically expand a location path with descendant steps into a number of similar paths using the child axis only.

Incorporating data-guides in our structure-aware query processing requires significant extensions of our data structures. Analogously to the tag-term index that includes the pre/postorder labeling scheme, we now index and query for bucketid-term pairs as the main building blocks for our query processing strategies. This *bucketid-term index* contains, for each bucketid-term pair, entries of the form
(bucketid, term, docid, pre, post, level, score, maxscore)
that are sorted in the same block structure as the tag-term index. We also maintain a *bucket index* that corresponds to the structure index and contains, for each bucket id, entries of the form
(bucketid, docid, pre, post, level).
The data-guide and all its distinct path-to-bucketid mappings can typically be kept in-memory for the type of document collections we investigate; into main memory when the engine starts. The memory consumption of the data-guide is typically negligible; for the INEX IEEE collection the data-guide has about 10,000 distinct path entries.

Each content condition in the query now opens a sequential scan on this index. All assumptions on random accesses for content and navigational conditions follow analogously to the pre/postorder labeling scheme. Using bucketid-term pairs for querying only provides a structural filter for element contents, since the paths do not provide unique identifiers for the elements as required for joining their scores. In particular, evaluating branching path queries only on the basis of data-guides would make us run into the danger of returning false positives. Therefore, structural joins are furthermore performed on the pre-/postorder labels in the form of a *hybrid index* which serves two purposes:

1) We use data-guides for query rewriting only, and encode whole paths into a compact bucket id with lower selectivity than simple tags.
2) We perform structural joins on pre-/postorder labels, and thus are able to reuse our efficient join algorithm and implementation.

The latter point enables the query rewriter to dynamically select the most appropriate index structure for individual query nodes and to efficiently process mixed query conditions, with some navigational conditions referring to data-guide locators and some using individual tag conditions.

Figure 16 depicts the approach for the example location path `//article//sec//par` that is merged into a content condition with the term "database". Let us assume that the data-guide lookup yields only three different matching paths with respect to the child axis, namely for `/article/sec/par`, `/article/sec/ss1/par`, and `/article/sec/ss2/par`. Note that it is also possible to incorporate path similarities at this point, e.g., along the lines of [81,82], as indicated by the figure. An Incremental Merge operator is used to determine the order in which inverted index lists for the respective bucket-id-term pairs are merged, again merging whole element blocks and propagating them for the structural joins with other element blocks at different query dimensions for each candidate.

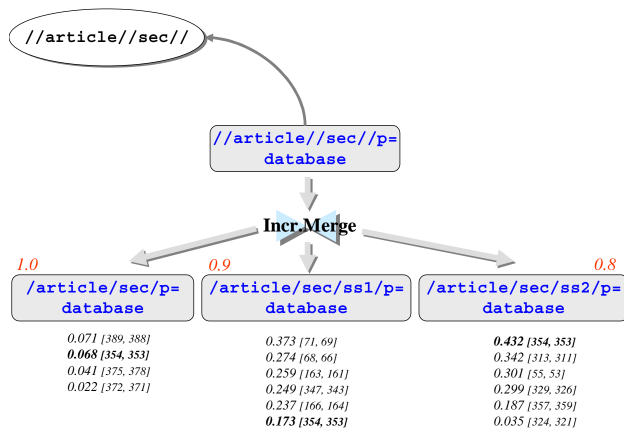

Fig. 16 Dynamic expansion of the descendant axis for a data-guide-like location path.

The query rewriter can incrementally query the data-guide for all path prefixes and break up the location path into a tag sequence (thus switching from data-guides to the pre/postorder scheme) as soon as the amount of distinct bucket ids for the path prefix exceeds a certain threshold value. The exact
choice on when to keep a location path with descendant steps for being processed with a data-guide, and when to split the path into a sequence of single navigational tags is collection-dependent. Initializing a huge amount of database cursors for the Incremental Merge algorithm may become more expensive than the actual query execution; we found a threshold of 12–24 a good choice for the INEX IEEE collection. Although we do not consider data-guides to be a panacea for addressing lowly selective structure (with few matches), dynamically switching between data-guides and tag-term pairs in fact allows us to efficiently cover a broad range of XML data collections with different structural characteristics.

Note that supporting data-guides and tag-term pairs *simultaneously* in our inverted block-index organization is space-consuming, since it roughly doubles the index size. The decision on whether to index a collection using only data-guides or only tag-term pairs depends on the amount of variations of paths in the collection, thus considering the different salient properties of each index structure. Note that the compact in-memory data-guide may be further kept for filtering invalid edges in the pre/postorder mode, too, preventing the algorithm from performing unnecessary random lookups for generally unsatisfiable structural constraints.

10 Implementation

10.1 Database-Backed Index Implementation

TopX uses a relational DBMS as a storage backend. In the following, we discuss the schema setup using *Oracle 10g* with the option of leveraging space-efficient *Index Only Tables* (IOTs) [57] and the *index key compression* feature for our primary storage structures; all schema definitions can be transfered analogously to other DBMS's or file managers.

The tag-term index is implemented by an IOT with attributes concatenated in the order *(tag, term, maxscore, docid, score, pre, post, level)*. This is directly used for efficient sequential scanning of index lists for tag-term pairs. To also enable efficient random access to the tag-term index, we have created a B^+-tree index over the complete range of attributes in this table in the order *(docid, tag, term, score, maxscore, pre, post, level)*. By keeping all query-relevant attributes redundantly in the index (and thus forcing a full replication of the data), we prevent the DBMS from performing more expensive index-access-per-rowid plans (i.e., hidden random accesses between the index and the base table).

The structure index is implemented as another IOT with attributes concatenated in the order *(docid, tag, pre, post, level)*. There are similar database tables for the hybrid indexes discussed in Section 9.

10.2 Multi-threaded Query Processing

The general TopX architecture comprises a three-tier, multi-threaded hierarchy consisting of

1) the *main thread* that periodically maintains the data structure for the candidate bookkeeping and optionally updates the probabilistic predictors for candidate pruning and the adaptive scheduling decisions after each batch of b sorted index accesses,
2) the *scan threads* that iteratively read and join input tuples on top of the list buffers for a batch of b sorted accesses, and
3) the *buffer threads* that continuously refill a small buffer cache and control the actual disk I/O for each index list.

This three-level architecture builds on the observation that candidate pruning and scheduling decisions incur overhead and should be done only periodically, and joining and evaluating score bounds for candidate may incur high CPU load (in particular for path query evaluations), whereas the actual sequential index accesses are not critical in terms of CPU load.

To optimize query execution time, we need to ensure *continuous* and *asynchronous* disk operations throughout the whole query processing. With the above strategy of dividing index scans and candidate pruning into different threads, disk operations might get temporarily interrupted at the synchronization points, namely when all scan threads are suspended and the main thread is active with pruning. Therefore, apart from the result set prefetching at the database connector (e.g., ODBC or JDBC) or disk caching effects (which we cannot easily control), we add an additional small buffer for each physically stored index list that does not exceed the default batch size that is initially scheduled for the first round of round-robin-like index list accesses (e.g., a maximum of 1,000 tuples). We add an additional tier of buffer threads responsible for the actual disk reads and buffered index lists lookups to completely decouple the physical I/O performance from the query processing.

Then all scan threads solely work on top of these buffers which are constantly refilled by the tier of decoupled buffer threads with asynchronous disk I/O until the query processing terminates. This way, we experience no startup delays after notifying the scan thread which makes multi-threaded scheduling with different batch sizes per thread feasible, because the disk operations are not interrupted. The actual buffer threads are suspended, too, when the intermediate read buffer is filled to the maximum value, and they are notified when the buffer falls below some minimum fill threshold (e.g., a minimum of 100 tuples). The maximum buffer sizes may be chosen proportionally to the size b of the scheduled batches. Note that random accesses are triggered by the main thread that directly accesses the inverted lists. This type of access greatly benefits from the internal page-caching strategy of the underlying DBMS.

Figures 17 and 18 demonstrate the advantages of the multi-threading architecture in two small experiments, both conducting a batch of 50 TREC 2003 Web track queries for the topic distillation task on the GOV collection [34], but on two different hardware configurations. Figure 17 shows a wallclock run-time of 35.2 seconds for both the multi- and single threaded configuration (the latter scheduling SA batches in a simple round-robin style) using a single-CPU notebook (with a 1.6 GHz Centrino CPU) connected to an Oracle server via a 1 Gigabit LAN. This demonstrates the I/O boundedness of the algorithm for this particular configuration which is exactly what we would expect. The situation changes, however, when queries are executed directly on the server machine that also hosts the Oracle database (with a 3 GHz dual Xeon CPU) and tuples are read directly from the RAID disks. Figure 18 shows that the wallclock run-time significantly drops from 20.9 seconds in single-threaded mode to 8.3 seconds in multi-threaded mode which demonstrates that a single thread cannot exhaust the full I/O bandwidth on the server and the algorithm becomes CPU bounded. So multi-threading is a crucial performance issue, in particular on multi-CPU machines.

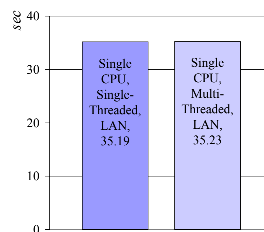

Fig. 17 Multi-threading vs. single-threading on a single CPU system.

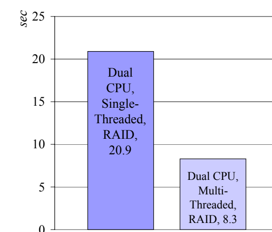

Fig. 18 Multi-threading vs. single-threading on a dual CPU system.

This way, the scan threads are totally decoupled from each other. Synchronization (object locking) for shared data structures only takes place when a candidate is pulled from the cache and the queue is updated, or when (occasionally) a candidate is found to be promoted into the top-k queue which happens much less frequently than updates on the candidate queue. A particularly nice feature of this architecture is that the more CPU-intensive XPath evaluations (see Section 5.3) can easily be made truly parallel at the level of the scan threads, because they just work concurrently on different documents.

11 Experiments

11.1 Setup & Data Collections

We focus our experiments with textual data on the TREC Terabyte collection which is the largest currently available text corpus with relevance assessments, consisting of about 25 million documents with a size of roughly 425 GB and 50 queries from the 2005 Terabyte Ad-Hoc task [94].

For XML on the other hand, we chose the INEX IEEE collection consisting of roughly 17,000 XML-ified CS journal articles and the 6 GB INEX Wikipedia [35] collection with about 660,000 XML-ified Wikipedia articles, yielding more than 130M elements and the respective batch of the 125 INEX 2006 Ad-Hoc queries. We also provide comparative studies from the official results of the INEX 2005 and 2006 benchmarks. Table 1 summarizes these collection statistics. In terms of bytes sizes and number of tuples contained in the inverted index (denoted by #Features in Table 1), the new INEX Wikipedia corpus is an order of magnitude larger than the previous INEX IEEE collection.

	#Docs	#Elmts.	#Featrs.	Size
TREC-TB	25,150,527	n/a	2,938 M	426 GB
INEX-IEEE	16,819	18 M	142 M	743 MB
INEX-Wiki	659,204	131 M	632 M	6.5 GB

Table 1 Source data sizes of the test collection used.

On a mainstream server machine with a dual XEON-3000 CPU, 4GB of RAM, and a large SCSI RAID-5, indexing these collections took between 280 minutes for INEX-Wiki and 14 hours for Terabyte, including stemming, stopword removal and computing the BM25-based scores. The materialization of the B^+-indexes required roughly the same amount of time as it included sorting a large intermediate table.

11.2 Evaluation Metrics

As for efficiency, we consider abstract query execution costs

$$cost := \#SA + c_R/c_S\ \#RA$$

i.e., a weighted sum of the number of tuples read through sorted and random accesses from our disk-resident index structures, as our primary metric analogously to [39]. The cost ratio c_R/c_S of a single sorted over a single random access has been determined to optimize our run-time figures at a value of 150 which nicely reflects our setup using Oracle as backend and JDBC as connector, with a relatively low sequential throughput but good random access performance because of the caching capabilities of the DBMS.

As for effectiveness, we refer to the relative and absolute precision values, as well as the non-interpolated mean average precision (MAP) [24,97] which displays the absolute (i.e., user-perceived) precision as a function of the absolute recall, using official relevance assessments provided by TREC or INEX. Furthermore, the following, more sophisticated and XML-specific metrics were newly introduced for the INEX benchmark 2005 [62]:

– *nxCG* – The normalized extended Cumulated Gain metrics is an extension of the cumulated gain (CG) metrics which aims to consider the dependency of XML elements (e.g., overlap and near-misses) within the evaluation.
– *ep/gr* – The expected-precision/gain-recall metric finally aims to display the amount of relative effort (where effort is measured in terms of the number of visited ranks) that the user is required to spend when scanning a system's result ranking. This effort is compared to the effort an ideal ranking would take in order to reach a given level of gain relative to the total gain that can be obtained.

Wallclock run-times were generally good but much more sustainable to these very caching effects, with average CPU run-times per query being in the order of 0.3 seconds for Wikipedia and 1.2 for Terabyte, and wallclock run-times being 3.4 and 6.2 seconds, respectively. All the reported cost figures are sums for the whole batch of benchmark queries, whereas the precision figures are macro-averaged. Altogether, the various algorithmic variants and pruning strategies described before open a huge variety of possible experiments and setups; and the following runs can merely try to provide a comprehensive overview over our most meaningful results.

11.3 Terabyte Runs

11.3.1 Baseline Top-k Competitors & Scheduling Options

We start with an overview on text data comparing the TopX with the most prominent variants of the TA-family of algorithms [39] such as TA, NRA, and CA, and a DBMS-style full-merge algorithm. Figure 19 presents the average cost savings of our extended scheduling strategy (Ben-Probing) which outperforms all our three top-*k* baseline algorithms by factors of in between 25 and 350 percent.

The DBMS-style merge join first joins all documents in the query-relevant index lists by their id and then sorts the joined tuples for reporting the final top-*k* results (eventually

using a partial sort). For $k = 10$, the non-approximate TopX run with the conservative pruning already outperforms this full-merge by a factor of 550 percent, while incurring query costs of about 9,323,012 compared to 54,698,963 for the full-merge. Furthermore, we are able to maintain this good performance over for a very broad range of k; only queries of considerably more than 1,000 requested results would make our algorithm degenerate over the full merge approach. The approximate TopX with a relatively low probabilistic pruning threshold of $\varepsilon = 0.1$ generally performs about 10–20 percent lower execution costs than the exact TopX setup which conforms exactly to the pruning behavior we would expect and the probabilistic guarantees for the result quality we provide in Section 7.

Even for $k = 1,000$, there is a 30 percent improvement over the best remaining baselines for these large values of k, namely full-merge and NRA which itself has almost converged to full-merge. Note that the end user of top-k results (as in Web search) would typically set k to 10–100, whereas application classes with automated result post-processing (such as multimedia retrieval) may choose k values between 100 and 1,000. Especially remarkable is the fact that we consistently approach the absolute lower bound for this family of algorithms (see also [18]) by about 20 percent even for large k, whereas both the CA and NRA baselines increasingly degenerate; CA even exceeds the full-merge baseline in terms of access cost for $k > 500$ (as it needs to scan most of the lists, just like the full-merge baseline, to their end, but additionally performs expensive random accesses). For $\varepsilon = 0.1$, we already touch the lower bound with hardly any loss in result precision (see also Figure 20).

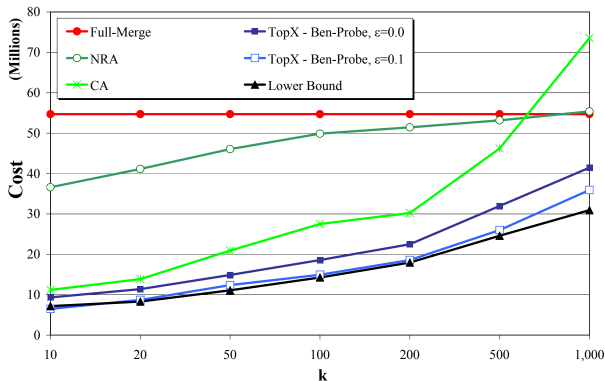

Fig. 19 Execution costs for TopX compared to various top-k baselines and a computed lower bound on Terabyte, for varying k.

Note that we measured the average query cost for the original TA algorithm with full random lookups for each candidate with a value of 154,188,975 already for $k = 10$, which could not even be plotted on the same scale as the other variants for the default cost ratio $c_R/c_S = 150$ (but also for lower ratios).

11.3.2 Pruning Effectiveness on Text

TopX yields a comparably good effectiveness on the Terabyte topics, with a peak mean average precision (MAP) of 0.19 and a peak precision@10 of 0.5, given that we use a standard BM25-based scoring function for text data. Figure 20 investigates the detailed probabilistic pruning behavior of TopX for the full range of $0 \leq \varepsilon \leq 1$ for a fixed value of $k = 10$, with $\varepsilon = 1.0$ (i.e., the extreme case) meaning that we immediately stop query processing after the first batch of b sorted accesses. Since Terabyte is shipped with official relevance judgments, we are able to study the result quality for both the *relative precision* (i.e., the overlap between the approximate and the exact top-k) and the *absolute precision* (i.e., the fraction of results officially marked as relevant by a human user for a particular topic), as well as MAP. Note that the recall-dependent MAP values are inherently low for $k = 10$.

We also see that the relative precision drops much faster than the absolute precision which means that, although different documents are returned at the top-k ranks, they are mostly equally relevant from a user's perspective. Particularly remarkable is the fact that the

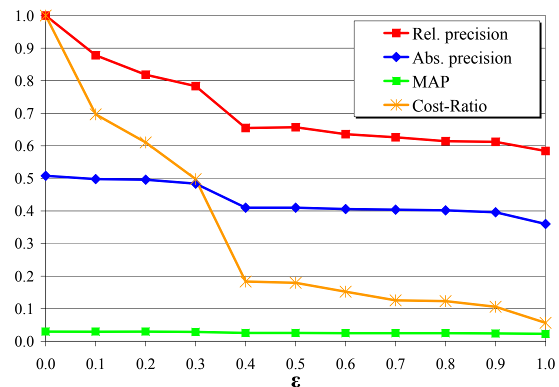

Fig. 20 Relative vs. absolute (i.e., user-perceived) retrieval precision and the cost-ratio as functions of ε on Terabyte, for $k = 10$.

$$cost\text{-}ratio(\varepsilon) := cost_{approx}(\varepsilon)/cost_{exact}$$

of the approximate TopX runs with probabilistic pruning over the cost of the exact TopX runs generally drops at a much faster rate than both the absolute and relative precision values. That is for $\varepsilon = 0.4$, we have less than 20 percent of the execution cost of the exact top-10, but we still achieve more than 65 percent relative precision (which confirms our probabilistic guarantees, see Section 7); and we even have

less than 10 percent loss in absolute precision according to the official relevance assessments on Terabyte. We observed this trend for all collections and query setups we considered so far.

11.4 INEX-IEEE Runs

11.4.1 XML-Top-k Competitors

In addition to the full-merge baseline, which is inspired by the Holistic Twig Join of [22, 59, 29] in the XML case, two state-of-the-art XML-Top-k competitors were evaluated:

- `StructIndex`, the algorithm developed in [61] which uses a structure index to preselect candidates that satisfy the path conditions and then uses a TA-style evaluation strategy with eager random access to compute the top-k result.
- `StructIndex`$^+$, an optimized version of the `StructIndex` top-k algorithm, using also the extent chaining technique of [61].

Figure 21 shows that already the conservative TopX method without probabilistic pruning ($\varepsilon = 0$) reduces execution costs by 300–500 percent. A detailed analysis shows that TopX reduces the number of expensive RA's even by an absolute factor of 50 to 80 (!) compared to the TA-based `StructIndex` competitors on INEX in both the `Min-Probe` and `Ben-Probe` configurations, with very good rates of inexpensive SA's. `StructIndex`$^+$ even exceeds `StructIndex` in terms of RA's and thus incurs fewer SA's than `StructIndex` or TopX but much higher overall execution costs. Both `StructIndex` and `StructIndex`$^+$ have higher cost than the full-merge baseline for large k, as they need to read large fractions of the lists and perform many RA's.

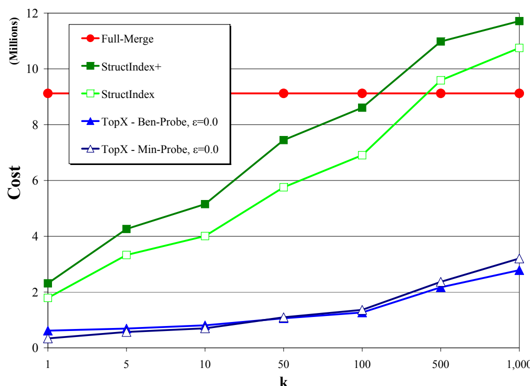

Fig. 21 Execution costs for TopX compared to the StructIndex and full-merge competitors on INEX-IEEE, for varying k.

Here, the simple Min-Probe scheduling even slightly outperforms Ben-Probe on INEX, in terms of saving random accesses to the navigational tags. For the XML case, the Ben-Probing was limited to scheduling RA's to the navigational tag conditions in the auxiliary table `TagsRA` only. A fully enabled Ben-Probe scheduling to content conditions could have further decreased the execution cost but was omitted here, because the Min-Probe competitor inherently cannot determine RA's to content conditions (see also [26, 18]). Recall that random accesses strongly affect running times, because they incur in a (empirically measured) run-time factor of about 150 in our specific hardware and database setup.

11.4.2 Pruning Effectiveness on XML

Figure 22 shows that again the relative precision value degrades at a much higher rate than the absolute result precision in terms of relevant elements found. This means that different results are returned at the top ranks, but they are equally good from a user perspective according to the official relevance assessments of INEX. Again, the recall-dependent MAP values are inherently low for $k = 10$; for $k = 1,000$ we achieved a remarkably good MAP value of 0.17.

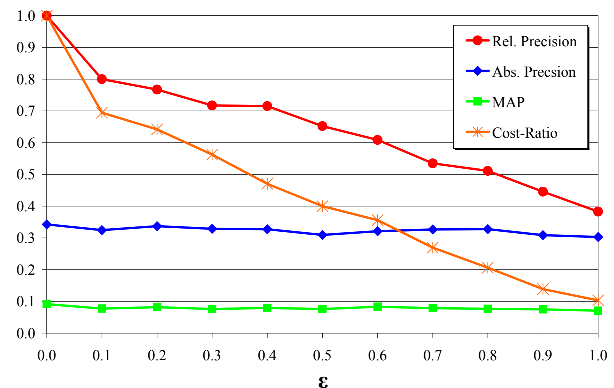

Fig. 22 Precision and Mean-Average-Precision (MAP) as functions of ε on INEX-IEEE, for $k = 10$.

11.4.3 Comparative Studies

For the INEX 2005 setting, the benchmark included a set of 40 keyword-only (CO) and a distinct set of 47 structural queries (CAS) with relevance assessments that were evaluated on the INEX-IEEE corpus. While the results for CO queries were reasonable with a peak position 19 out of 55 submitted runs for the generalized nxCG@10 metric (which is not surprising as we use a rather standard content scoring model), TopX performed very well for CAS queries, ranking among the top-5 of 25, with a peak position 1 for two of the five official evaluation methods. The two officially submitted TopX CAS runs finally ranked at position 1 and 2 out of 25 submitted runs for the strict nxCG@10 metric with a

very good value of 0.45 for both runs, and they still rank at position 1 and 6 for MAP with values of 0.0322 and 0.0272, respectively.

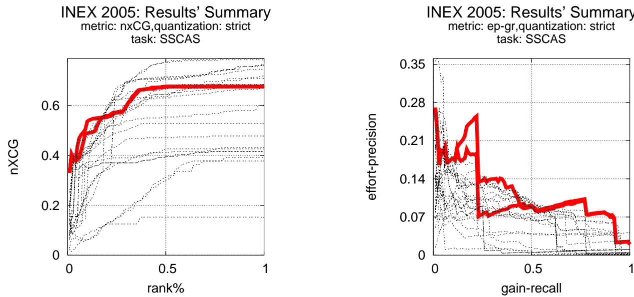

Fig. 23 Official INEX-IEEE '05 benchmark results of TopX compared to all participants (using the *ep-gr* and *nxCG* metrics).

Figure 23 shows the nxCG and ep/gr plots for the two CAS runs, one with and one without considering expensive text predicates which performed almost equally well in this case. We see that TopX quickly reaches a maximum in the cumulated gain measure at about 50 percent of the returned ranks and then saturates, which is an excellent property for a top-k engine, because the best results are typically detected and returned at the first ranks already. Particularly nice is also the high peak of the second TopX run in the ep/gr metric which makes this run stand out in comparison to its competitors.

11.5 INEX-Wikipedia Runs

As for Wikipedia, we provide a detailed comparison of the CO and CAS interpretations of the INEX queries. As opposed to the INEX-IEEE '05 setting, each of the 125 new INEX-Wikipedia '06 queries comes shipped in the CO and CAS flavors. The following runs aim to compare the different performance issues between CO and CAS for our system.

11.5.1 CO vs. CAS

Figure 24 shows that we generally observe similarly good performance trends as for Terabyte or IEEE, with cost-savings of a factor of up to 700 percent for CAS and 250 percent for CO when compared to full-merge, and the performance advantage remains very good even for large values of k, because we never need to scan the long (i.e., highly selective) element lists for the navigational query tags which has to be performed by any non-top-k-style algorithm, e.g., when using Holistic Twig joins.

Figure 26 depicts a detailed comparison of the query costs being split into individual sorted (#SA) and random

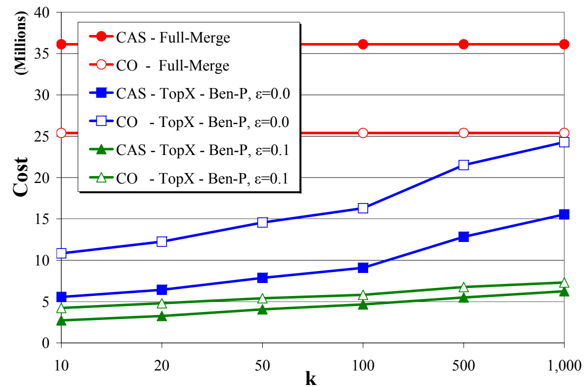

Fig. 24 Execution costs reported for the CAS and CO queries as functions of k on Wikipedia, for $\varepsilon = 0.0$ and $\varepsilon = 0.1$.

(#RA) index accesses for the CO and CAS flavors of the Wikipedia queries. It shows that we successfully limit the amount of RA to less than about 2 percent of the #SA according to our cost model. This ratio is maintained also in the case of structured data and queries which is a unique property among current XML-top-k engines.

11.5.2 Pruning Effectiveness for CO vs. CAS

Figure 25 demonstrates a similarly good pruning behavior of TopX for both the CO and CAS queries on Wikipedia, again showing a very good quality over run-time ratio for the probabilistic candidate pruning component. As expected, these runs confirm the fact that CAS queries are significantly less expensive than CO queries for TopX, because for the structure-constrained queries we largely benefit from the lowly selective, precomputed inverted lists over combined tag-term pairs. Note that official relevance judgments for the 2006 Wikipedia benchmark as required for computing the absolute precision figures are not yet available, such that absolute precision values are omitted here.

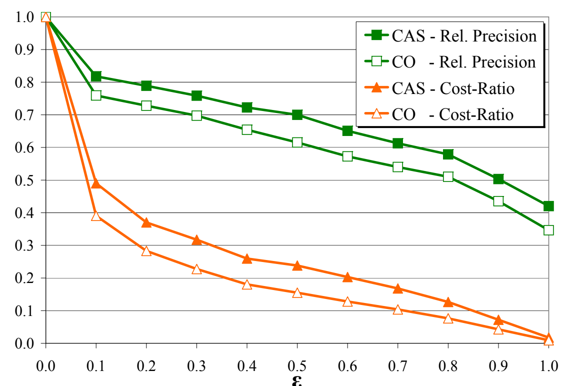

Fig. 25 Relative retrieval precision and cost-ratios for the CAS and CO queries as functions of ε on Wikipedia, for $k = 10$.

11.5.3 Expansion Efficiency for CO vs. CAS

Figure 27 finally shows an impressive run-time advantage for the dynamic query expansion approach compared to both full-merge and TopX when performing static expansions (measured for the CAS case). The reported numbers reflect large, automatic thesaurus expansions of the original Wikipedia queries based on WordNet, with up to $m = 292$ distinct query dimensions (i.e., keywords and phrases embedded into the structure of the CAS query).

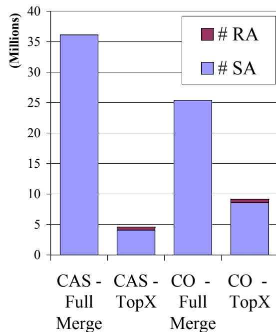

Fig. 26 #SA and #RA for full-merge vs. TopX, for $k = 10$ and $\varepsilon = 0$.

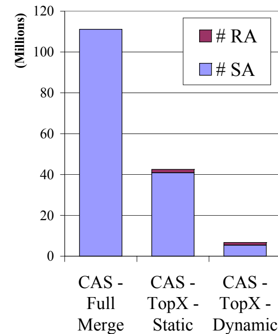

Fig. 27 #SA and #RA for full-merge vs. TopX with static and dynamic expansion, for $k = 10$ and $\varepsilon = 0$.

11.5.4 Comparative Studies

For the INEX 2006 setting, 114 out of the 125 initial queries come with human relevance assessments and were therefore used to compare different engines. There was no separate evaluation of the CAS queries and the strict evaluation mode was dropped, and hence the relative performance of TopX was comparable to 2005 with a rank of 24 out of 106 submitted runs for the generalized ep/gr metric. However, INEX 2006 introduced a new retrieval task, *AllInContext*, where a search engine first had to identify relevant articles (the fetching phase), and then identify the relevant elements within the fetched articles (the browsing phase). This task can be easily mapped to the TopX document evaluation mode, and in fact TopX performed extremely well with ranks of 1 and 2 (two TopX runs with different parameters) for the generalized precision/recall metric (gP) [63] at 50 results. Figure 28 shows the gP plots for the two runs (the two bold red lines at the top of the chart), one with and one without considering expensive text predicates which performed almost equally well in this case.

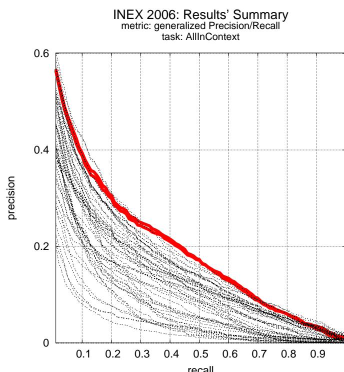

Fig. 28 Official INEX-Wikipedia '06 benchmark results of TopX compared to all participants for the AllInContext task (using the gP metric).

12 Conclusions and Future Work

TopX is an efficient and effective search engine for non-schematic XML documents, with the full functionality of XPath Full-Text and supporting also the entire range of text, semistructured, and structured data. It achieves effective ranked retrieval by means of an XML-specific extension of the probabilistic-IR BM25 relevance scoring model, and also leverages thesauri and ontologies for word-sense disambiguation and robust query expansion. It achieves scalability and efficient top-k query processing by means of extended threshold algorithms with specific priority-queue management, judicious scheduling of random accesses to index entries, and probabilistic score predictions for early pruning of top-k candidates.

TopX has been the official host used for the INEX 2006 topic development phase, and its Web Service interface has been used by the INEX 2006 Interactive Track. During the topic development phase, more than 10,000 CO and CAS queries from roughly 70 different participants were conducted partly in parallel sessions over the new Wikipedia XML index. The complete TopX indexing and query processing framework is available as open source code at the URL `http://topx.sourceforge.net`.

12.1 Lessons Learned

TopX uses a relational database system to manage its indexes, relying on 30 years of database research for efficient index structures, updates, and transaction management. In fact, this solution has turned out to be flexible and easy to use, especially for complex precomputations of scores and corpus statistics that require joins and aggregations, However, the price we had to pay for this convenience was high overhead compared to dedicated index data structures like inverted files, both in run-time and in storage space. Run-time suffered from the expensive interface crossing incurred by JDBC and access layers inside the database server. Customized inverted files, on the other hand, would provide more

light-weight storage management. Preliminary experiments with file-based storage have shown a speedup for sorted and random accesses of up to a factor of 20, while requiring an order of magnitude less storage space.

The self-throttling query expansion mechanism introduced in TopX is a major advantage over existing solutions, with respect to both effectiveness and efficiency. However, for robust query expansion, the quality of the underlying ontologies and thesauri is critical. So far, we have been mostly relying on WordNet, but we plan to use richer, high-quality ontologies [85] and we would also like to explore domain-specific ontologies.

12.2 Future Work

Our future work on this subject will focus on four major issues:

- We plan to extend the tree-oriented view of XML data into a graph-based data model that includes both intra-document references among elements and inter-document links given by XLink pointers or href-based hyperlinks. In terms of functionality, we can build on the initial work by [46] along these lines, but one of the major challenges is to ensure efficiency given the additional complexity of moving from trees to graphs.
- We want to address extended forms of scoring and score aggregation, most notably, proximity predicates. This may involve handling non-monotonous aggregation functions and will mandate significant extensions to our top-k search algorithms.
- We plan to reconsider our indexing methods and investigate specialized data structures for inverted lists, most notably, to exploit various forms of compression [104], efficient support for prefix matching [19], and special structures for speeding up phrase matching [99].
- We consider re-implementing core parts of the TopX engine in C++, replacing the relational database backend with our own storage system including a customized implementation of inverted block indexes. Notwithstanding the good performance of our current system, we would expect to gain another order of magnitude in run-time efficiency by this kind of specific and more mature engineering.

We believe that the integration of DB and IR functionalities and system architectures will remain a strategically important and rewarding research field, and we hope to make further contributions to this area.

References

1. Aboulnaga, A., Alameldeen, A.R., Naughton, J.F.: Estimating the selectivity of XML path expressions for internet scale applications. In: VLDB, pp. 591–600. Morgan Kaufmann (2001)
2. Agrawal, S., Chaudhuri, S., Das, G., Gionis, A.: Automated ranking of database query results. In: CIDR (2003)
3. Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: A primitive for efficient XML query pattern matching. In: ICDE, pp. 141–152. IEEE Computer Society (2002)
4. Al-Khalifa, S., Yu, C., Jagadish, H.V.: Querying structured text in an XML database. In: SIGMOD, pp. 4–15. ACM (2003)
5. Amato, G., Rabitti, F., Savino, P., Zezula, P.: Region proximity in metric spaces and its use for approximate similarity search. ACM Trans. Inf. Syst. **21**(2), 192–227 (2003)
6. Amer-Yahia, S., Botev, C., Dörre, J., Shanmugasundaram, J.: XQuery Full-Text extensions explained. IBM Systems Journal **45**:2, 335–352 (2006)
7. Amer-Yahia, S., Botev, C., Shanmugasundaram, J.: TeXQuery: a full-text search extension to XQuery. In: WWW, pp. 583–594. ACM (2004)
8. Amer-Yahia, S., Case, P., Rölleke, T., Shanmugasundaram, J., Weikum, G.: Report on the DB/IR panel at SIGMOD 2005. SIGMOD Record **34**(4), 71–74 (2005)
9. Amer-Yahia, S., Curtmola, E., Deutsch, A.: Flexible and efficient XML search with complex full-text predicates. In: SIGMOD, pp. 575–586. ACM (2006)
10. Amer-Yahia, S., Koudas, N., Marian, A., Srivastava, D., Toman, D.: Structure and Content Scoring for XML. In: VLDB, pp. 361–372. ACM (2005).
11. Amer-Yahia, S., Lakshmanan, L.V.S., Pandit, S.: FleXPath: Flexible structure and full-text querying for XML. In: SIGMOD, pp. 83–94. ACM (2004)
12. Amer-Yahia, S., Lalmas, M.: XML Search: Languages, INEX and Scoring. SIGMOD Record **36**:7, 16–23 (2006)
13. Anh, V.N., de Kretser, O., Moffat, A.: Vector-space ranking with effective early termination. In: SIGIR, pp. 35–42. ACM (2001)
14. Anh, V.N., Moffat, A.: Impact transformation: effective and efficient web retrieval. In: SIGIR, pp. 3-10. ACM (2002)
15. Anh, V.N., Moffat, A.: Pruned query evaluation using pre-computed impacts. In: SIGIR, pp. 372–379. ACM (2006)
16. Avnur, R., Hellerstein, J.M.: Eddies: Continuously adaptive query processing. In: SIGMOD, pp. 261–272. ACM (2000)
17. Baeza-Yates, R.A., Ribeiro-Neto, B.A.: Modern Information Retrieval. ACM Press / Addison-Wesley (1999)
18. Bast, H., Majumdar, D., Theobald, M., Schenkel, R., Weikum, G.: IO-Top-k: Index-optimized top-k query processing. In: VLDB, pp. 475–486. ACM (2006)
19. Bast, H., Weber, I.: Type less, find more: fast autocompletion search with a succinct index. In: SIGIR, pp. 364–371. ACM (2006)
20. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. **33**(3), 322–373 (2001)
21. Bruno, N., Chaudhuri, S., Gravano, L.: Top-k selection queries over relational databases: Mapping strategies and performance evaluation. ACM Trans. Database Syst. **27**(2), 153–187 (2002)
22. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD, pp. 310–321. ACM (2002)
23. Buckley, C., Lewit, A.F.: Optimization of inverted vector searches. In: SIGIR, pp. 97–110. ACM (1985)
24. Buckley, C., Voorhees, E.M.: Evaluating evaluation measure stability. In: SIGIR, pp. 33–40. ACM (2000)
25. Carmel, D., Maarek, Y.S., Mandelbrod, M., Mass, Y., Soffer, A.: Searching XML documents via XML fragments. In: SIGIR, pp. 151–158. ACM (2003)
26. Chang, K.C.C., Hwang, S.: Minimal probing: supporting expensive predicates for top-k queries. In: SIGMOD, pp. 346–357. ACM (2002)

27. Chaudhuri, S., Gravano, L., Marian, A.: Optimizing top-k selection queries over multimedia repositories. IEEE Trans. Knowl. Data Eng. **16**(8), 992–1009 (2004)
28. Chinenyanga, T.T., Kushmerick, N.: Expressive retrieval from XML documents. In: SIGIR, pp. 163–171. ACM (2001)
29. Choi, B., Mahoui, M., Wood, D.: On the optimality of holistic algorithms for twig queries. In: DEXA, *Lecture Notes in Computer Science*, vol. 2736, pp. 28–37. Springer (2003)
30. Ciaccia, P., Patella, M.: Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In: ICDE, pp. 244–255. IEEE Computer Society (2000)
31. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: A semantic search engine for XML. In: VLDB, pp. 45–56. Morgan Kaufmann (2003)
32. Consens, M.P., Baeza-Yates, R.A.: Database and Information Retrieval Techniques for XML. In: ASIAN, pp. 22–27. Springer (2005)
33. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Clifford, S.: Introduction of Algorithms. The MIT Press (2001)
34. Craswell, N., Hawking, D., Wilkinson, R., Wu, M.: Overview of the TREC 2003 Web track. In: TREC, pp. 78–92 (2003)
35. Denoyer, L., Gallinari, P.: The Wikipedia XML Corpus. SIGIR Forum (2006)
36. Donjerkovic, D., Ramakrishnan, R.: Probabilistic optimization of top n queries. In: VLDB, pp. 411–422. Morgan Kaufmann (1999)
37. Fagin, R.: Combining fuzzy information from multiple systems. J. Comput. Syst. Sci. **58**(1), 83–99 (1999)
38. Fagin, R.: Combining fuzzy information: an overview. SIGMOD Record **31**(2), 109–118 (2002)
39. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS, pp. 102–113. ACM (2001)
40. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci. **66**(4), 614–656 (2003)
41. Fegaras, L.: XQuery processing with relevance ranking. In: XSym, *Lecture Notes in Computer Science*, vol. 3186, pp. 51–65. Springer (2004)
42. Fellbaum, C. (ed.): WordNet: An Electronic Lexical Database. MIT Press (1998)
43. Fuhr, N., Großjohann, K.: XIRQL: A query language for Information Retrieval in XML documents. In: SIGIR, pp. 172–180. ACM (2001)
44. Goldman, R., Widom, J.: Dataguides: Enabling query formulation and optimization in semistructured databases. In: VLDB, pp. 436–445. Morgan Kaufmann (1997)
45. Grabs, T., Schek, H.J.: PowerDB-XML: Scalable XML processing with a database cluster. In: Intelligent Search on XML Data, pp. 193–206 (2003)
46. Graupmann, J., Schenkel, R., Weikum, G.: The spheresearch engine for unified ranked retrieval of heterogeneous XML and web documents. In: VLDB, pp. 529–540. ACM (2005)
47. Grust, T.: Accelerating XPath location steps. In: SIGMOD, pp. 109–120. ACM (2002)
48. Grust, T., van Keulen, M., Teubner, J.: Staircase join: Teach a relational DBMS to watch its (axis) steps. In: VLDB, pp. 524–525. Morgan Kaufmann (2003)
49. Güntzer, U., Balke, W.T., Kießling, W.: Optimizing multi-feature queries for image databases. In: VLDB, pp. 419–428. Morgan Kaufmann (2000)
50. Güntzer, U., Balke, W.T., Kießling, W.: Towards efficient multi-feature queries in heterogeneous environments. In: ITCC, pp. 622–628. IEEE Computer Society (2001)
51. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRank: Ranked keyword search over XML documents. In: SIGMOD, pp. 16–27. ACM (2003)
52. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Trans. Database Syst. **24**(2), 265–318 (1999)
53. Hjaltason, G.R., Samet, H.: Index-driven similarity search in metric spaces. ACM Trans. Database Syst. **28**(4), 517–580 (2003)
54. Hristidis, V., Papakonstantinou, Y., Balmin, A.: Keyword proximity search on XML graphs. In: ICDE, pp. 367–378. IEEE Computer Society (2003)
55. Hung, E., Deng, Y., Subrahmanian, V.S.: TOSS: An extension of TAX with ontologies and similarity queries. In: SIGMOD, pp. 719–730. ACM (2004)
56. INitiative for the Evaluation of XML Retrieval (INEX). `http://inex.is.informatik.uni-duisburg.de`
57. Index-Organized Tables – Oracle9*i* Data Sheet. `http://www.oracle.com/technology/products/oracle9i/datasheets/iots/iot_ds.html`
58. Jagadish, H.V., Lakshmanan, L.V.S., Srivastava, D., Thompson, K.: TAX: A tree algebra for XML. In: DBPL, *Lecture Notes in Computer Science*, vol. 2397, pp. 149–164. Springer (2001)
59. Jiang, H., Wang, W., Lu, H., Yu, J.X.: Holistic twig joins on indexed XML documents. In: VLDB, pp. 273–284. Morgan Kaufmann (2003)
60. Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering indexes for branching path queries. In: SIGMOD, pp. 133–144. ACM (2002)
61. Kaushik, R., Krishnamurthy, R., Naughton, J.F., Ramakrishnan, R.: On the integration of structure indexes and inverted lists. In: SIGMOD, pp. 779–790. ACM (2004)
62. Kazai, G., Lalmas, M.: INEX 2005 evaluation measures. In: 4th International Workshop of the Initiative for the Evaluation of XML Retrieval, *Lecture Notes in Computer Science*, vol. 3977, pp. 16–29. Springer (2005)
63. Lalmas, M., Kazai, G., Kamps, J., Pehcevski, J., Piwowarski, B., Robertson, S.: INEX 2006 Evaluation Measures. In: 5th International Workshop of the Initiative for the Evaluation of XML Retrieval, *Lecture Notes in Computer Science*, vol. 4518. Springer (2007)
64. Li, Y., Yu, C., Jagadish, H.V.: Schema-free XQuery. In: VLDB, pp. 72–83. Morgan Kaufmann (2004)
65. Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., Parr, R.: XPath-Learner: An on-line self-tuning Markov histogram for XML path selectivity estimation. In: VLDB, pp. 442–453. Morgan Kaufmann (2002)
66. List, J.A., Mihajlovic, V., Ramírez, G., de Vries, A.P., Hiemstra, D., Blok, H.E.: TIJAH: Embracing IR Methods in XML Databases. Inf. Retr. **8**(4), 547–570 (2005)
67. Liu, S., Chu, W.W., Shahinian, R.: Vague Content and Structure (VCAS) Retrieval for Document-centric XML Collections. In: WebDB, pp. 79–84 (2005)
68. Marian, A., Amer-Yahia, S., Koudas, N., Srivastava, D.: Adaptive processing of top-k queries in XML. In: ICDE, pp. 162–173. IEEE Computer Society (2005)
69. Marian, A., Bruno, N., Gravano, L.: Evaluating top-k queries over Web-accessible databases. ACM Trans. Database Syst. **29**(2), 319–362 (2004)
70. Mass, Y., Mandelbrod, M.: Component Ranking and Automatic Query Refinement for XML Retrieval. In: 3rd International Workshop of the INitiative for the Evaluation of XML Retrieval, *Lecture Notes in Computer Science*, vol. 3493, pp. 73–84. Springer (2004)
71. Moffat, A., Zobel, J.: Self-Indexing Inverted Files for Fast Text Retrieval. ACM Trans. Inf. Syst. **14**(4), 349–379 (1996)
72. Natsev, A., Chang, Y.C., Smith, J.R., Li, C.S., Vitter, J.S.: Supporting incremental join queries on ranked inputs. In: VLDB, pp. 281–290. Morgan Kaufmann (2001)
73. Nepal, S., Ramakrishna, M.V.: Query processing issues in image (multimedia) databases. In: ICDE, pp. 22–29. IEEE Computer Society (1999)
74. Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrieval with frequency-sorted indexes. JASIS **47**(10), 749–764 (1996)

75. Polyzotis, N., Garofalakis, M.N., Ioannidis, Y.E.: Approximate XML query answers. In: SIGMOD, pp. 263–274. ACM (2004)
76. Polyzotis, N., Garofalakis, M.N.: XSKETCH synopses for XML data graphs. ACM Trans. Database Syst. **31**:3, 1014–1063 (2006)
77. Reid, J., Lalmas, M., Finesilver, K., Hertzum, M.: Best entry points for structured document retrieval (Part I and II). Inf. Process. Manage. **42**:1, 74–105 (2006)
78. Robertson, S.E., Spärck-Jones, K.: Relevance weighting of search terms. Journal of the American Society for Information Science **27**(1), 129–146 (1976)
79. Robertson, S.E., Walker, S.: Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In: SIGIR, pp. 232–241. ACM/Springer (1994)
80. Rocchio Jr., J.: Relevance feedback in Information Retrieval. In: G. Salton (ed.) The SMART Retrieval System: Experiments in Automatic Document Processing, chap. 14, pp. 313–323. Prentice Hall (1971)
81. Schenkel, R., Theobald, M.: Feedback-driven structural query expansion for ranked retrieval of XML data. In: EDBT, pp. 331–348. Springer (2006)
82. Schenkel, R., Theobald, M.: Structural feedback for keyword-based XML retrieval. In: ECIR, pp. 326–337. Springer (2006)
83. Schlieder, T., Meuss, H.: Querying and ranking XML documents. JASIST **53**(6), 489–503 (2002)
84. Soffer, A., Carmel, D., Cohen, D., Fagin, R., Farchi, E., Herscovici, M., Maarek, Y.S.: Static index pruning for information retrieval systems. In SIGIR, pp. 43–50. ACM (2001).
85. Suchanek, F., Kasneci, G., Weikum, G.: YAGO: A core of semantic knowledge unifying WordNet and Wikipedia. In: WWW (2007).
86. Tao, Y., Faloutsos, C., Papadias, D.: The Power-method: a comprehensive estimation technique for multi-dimensional queries. In: CIKM, pp. 83–90 (2003)
87. Theobald, A., Weikum, G.: Adding relevance to XML. In: WebDB (Informal Proceedings), pp. 35–40 (2000)
88. Theobald, A., Weikum, G.: The index-based XXL Search Engine for querying XML data with relevance ranking. In: EDBT, pp. 477–495. Springer (2002)
89. Theobald, M., Schenkel, R., Weikum, G.: Exploiting structure, annotation, and ontological knowledge for automatic classification of XML data. In: WebDB, pp. 1–6 (2003)
90. Theobald, M., Schenkel, R., Weikum, G.: Efficient and self-tuning incremental query expansion for top-*k* query processing. In: SIGIR, pp. 242–249. ACM (2005)
91. Theobald, M., Schenkel, R., Weikum, G.: An efficient and versatile query engine for TopX search. In: VLDB, pp. 625–636. ACM (2005)
92. Theobald, M., Schenkel, R., Weikum, G.: The TopX DB & IR Engine. In: SIGMOD. ACM (2007)
93. Theobald, M., Weikum, G., Schenkel, R.: Top-*k* query evaluation with probabilistic guarantees. In: VLDB, pp. 648–659. Morgan Kaufmann (2004)
94. Text REtrieval Conference (TREC). `http://trec.nist.gov/`
95. Trotman, A., Sigurbjörnsson, B.: Narrowed Extended XPath I (NEXI). In: 3rd International Workshop of the INitiative for the Evaluation of XML Retrieval, *Lecture Notes in Computer Science*, vol. 3493, pp. 16–40. Springer (2004)
96. Vagena, Z., Moro, M.M., Tsotras, V.J.: Twig query processing over graph-structured XML data. In: WebDB, pp. 43–48 (2004)
97. Vorhees, E.: Overview of the TREC 2004 Robust retrieval track. In: TREC, pp. 69–77 (2004)
98. de Vries, A.P., Mamoulis, N., Nes, N., Kersten, M.L.: Efficient *k*-nn search on vertically decomposed data. In: SIGMOD, pp. 322–333. ACM (2002)
99. Williams, H.E., Zobel, J., Bahle, D.: Fast phrase querying with combined indexes. ACM Trans. Inf. Syst. **22**:4, 573–594 (2004)
100. XQuery 1.0 and XPath 2.0 Full-Text. `http://www.w3.org/TR/xquery-full-text/`
101. Wu, Y., Patel, J.M., Jagadish, H.V.: Structural join order selection for XML query optimization. In: ICDE, pp. 443–454. IEEE Computer Society (2003)
102. Yu, C.T., Sharma, P., Meng, W., Qin, Y.: Database selection for processing k nearest neighbors queries in distributed environments. In: JCDL, pp. 215–222. ACM (2001)
103. Zhang, C., Naughton, J.F., DeWitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: SIGMOD, pp. 425–436. ACM (2001)
104. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**:2 (2006)