TopX and XXL at INEX 2005

Martin Theobald, Ralf Schenkel, and Gerhard Weikum

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{mtb, schenkel, weikum}@mpi-inf.mpg.de

Abstract. We participated with two different and independent search engines in this year's INEX round: The XXL Search Engine and the TopX engine. As this is the first participation for TopX, this paper focuses on the design principles, scoring, query evaluation and results of TopX. We shortly discuss the results with XXL afterwards.

1 TopX – System Overview

Our query processing methods are based on precomputed index lists that are sorted in descending order of appropriately defined scores for individual tag-term content conditions, and our algorithmic rationale for top-k queries on XML data follows that of the family of *threshold algorithms (TA)* [2, 4, 5]. In order to find the top-k matches for multidimensional queries (e.g., with multiple content and structure conditions), scoring, and ranking them, TopX scans all relevant index lists in an interleaved manner. In each scan step, when the engine sees the score for a data item in one list, it combines this score with scores for the same data item previously seen in other index lists into a *global score* using a monotonic aggregation function such as weighted summation. We perform in-memory structural joins for content-and-structure (CAS) queries using pre-/postorder labels between whole element blocks for each query condition grouped by their document ids.

1.1 Top-k Query Processing for Semistructured Data

The query processor decomposes the query into *content conditions*, each of which refers to exactly one tag-term pair, and into additional elementary tag conditions (e.g., for navigation of branching path queries), plus the path conditions that constrain the way how the matches for the tag-term pairs and elementary tag conditions have to be connected. For NEXI, we concentrate on content conditions that refer to the descendant axis, i.e., the full-text contents of elements. This way, each term is connected to its last preceding tag in the location path, in order to merge each tag-term pair into a single query condition with a corresponding list in the precomputed inverted index. Note that sequential reads are performed for these content-related tag-term-pairs, only, whereas additional structural query conditions for element paths or branching path queries are performed through a few judiciously scheduled random lookups on a separate, more compact element table.

N. Fuhr et al. (Eds.): INEX 2005, LNCS 3977, pp. 282–295, 2006.

The rationale for these distinctions is that random accesses to disk resident index structure are typically one or two orders of magnitude more expensive than sorted accesses. Note that an index list (e.g., for a single term) on a large data collection may be very long, in the order of megabytes (i.e., multiple disk tracks), and the total index size may easily exceed a several Gigabytes so that only the "hottest" fragments (i.e., prefixes of frequently needed lists) can be cached in memory. Sorted access benefits from sequential disk I/O with asynchronous prefetching and high locality in the processor's cache hierarchy; so it has much lower amortized costs than random access. Threshold algorithms with eager random accesses look up the scores for a data item in all query-relevant index lists, when they first see the data item in one list. Thus, they can immediately compute the global score of the item, and need to keep only the current top-k items with their scores in memory. Algorithms with a focus on sorted access do not eagerly look up all candidates' global scores and therefore need to maintain a candidate pool in memory, where each candidate is a partially evaluated data item d that has been seen in at least one list and may qualify for the final top-k result based on the following information (we denote the score of data item d in the i-th index list by $s(t_i, d)$, and we assume for simplicity that the score aggregation is summation):

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

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

1.2 Periodic Queue Maintenance and Early Candidate Pruning

All intermediate candidates that are of potential relevance for the final top-k results are collected in a hash structure (the *cache*) in main memory; this data structure has the full information about elements, $worstscores$, $bestscores$, etc. In addition, two priority queues merely containing pointers to these cache entries are maintained in memory and periodically updated. The top-k queue uses $worstscores$ as priorities to organize the current top-k documents, and the candidate queue uses $bestscores$ as priorities to maintain the stopping condition for threshold termination.

Results from [11] show that only a small fraction of the top candidates actually has to be kept in the candidate queue to provide a proper threshold for algorithm termination. Since TopX typically stops before having scanned all the relevant index lists completely, much less candidates than the ones that occur in the inverted lists for a query have to be kept in the cache. Both queues contain disjoint subsets of items currently in the cache. If an item's $bestscore(d)$ drops below the current *min-k* threshold, it is dropped from the candidate queue as well as from the cache. The queue is implemented using a Fibonacci heap, with efficient amortized lookups and maintenance.

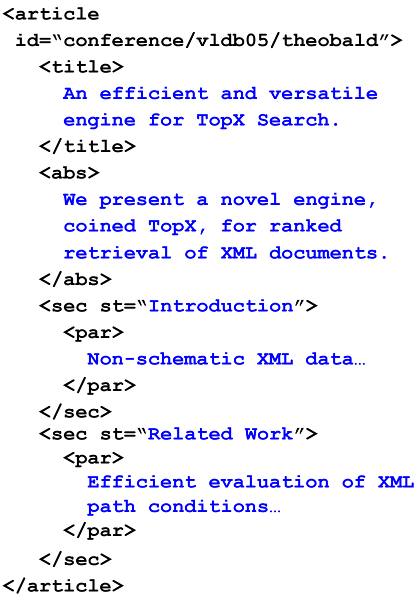
Optionally, TopX also supports various tunable probabilistic extensions to schedule random accesses for testing both content-related and structural query conditions as well as a probabilistic form of candidate pruning, thus yielding approximate top-k results with great run time gains compared to the conservative top-k baseline and probabilistic guarantees for the result quality [11]. However, for the current INEX experiments these probabilistic extensions were not employed, because here the focus is clearly on retrieval robustness rather than cutting edge performance.

2 Data and Scoring Model

2.1 Full-Content Indexing

We consider a simplified XML data model, where idref/XLink/XPointer links are disregarded. Thus every document forms a tree of nodes, each with a *tag* and a related *content*. We treat attributes nodes as children of the corresponding element node. The content of a node is either a text string or it is empty; typically (but not necessarily) non-leaf nodes have empty content.

With each node, we can additionally associate its *full-content* which is defined as the concatenation of the text contents of all the node's descendants. Optionally, we may apply standard IR techniques such as stemming and stop word removal to those text contents. This way, we conceptually treat each element as an eligible retrieval unit (i.e., in the classic IR notion of a document) with its expanded full-content text nodes as content, with no benchmark-specific tuning or preselection of commonly retrieved tags or the use of predefined retrieval units being necessary. In the following we focus on the descendant axis (i.e., the full-content case) as the much more important case for XML IR with vague search, thus following the NEXI specification; the case for the child axis follows analogously.



```
<article
 id="conference/vldb05/theobald">
   <title>
     An efficient and versatile
     engine for TopX Search.
   </title>
   <abs>
     We present a novel engine,
     coined TopX, for ranked
     retrieval of XML documents.
   </abs>
   <sec st="Introduction">
     <par>
       Non-schematic XML data…
     </par>
   </sec>
   <sec st="Related Work">
     <par>
       Efficient evaluation of XML
       path conditions…
     </par>
   </sec>
</article>
```

Fig. 1. An XML example

Figure 1 depicts an XML example snippet and Figure 2 illustrates our logical view of that document, with expanded full-content text nodes for each element.

For example, the full term frequency (*ftf*) of the term `xml` for the root element `article` has a value of 3 which reflects that the whole article element is definitely relevant for a query containing the term `xml`, however, it might be less compact than a more specific section or paragraph which should be taken into account by the scoring model.

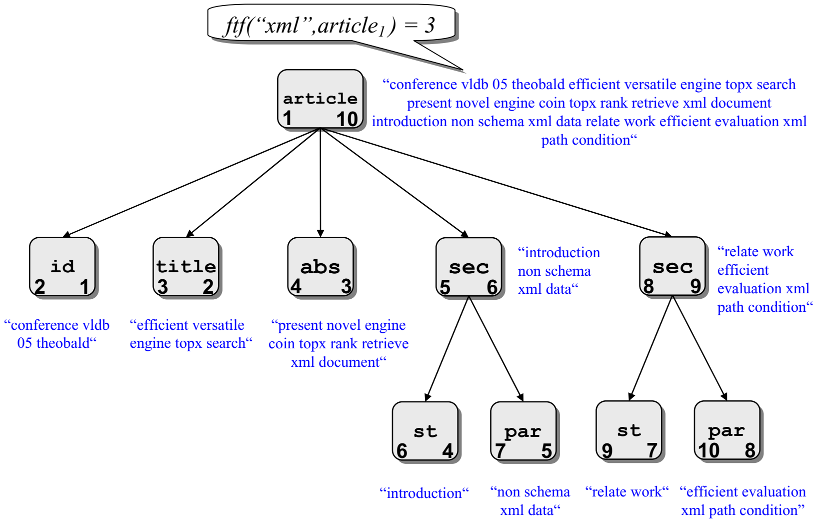

ftf("xml",article1) = 3
article 1 10
"conference vldb 05 theobald efficient versatile engine topx search present novel engine coin topx rank retrieve xml document introduction non schema xml data relate work efficient evaluation xml path condition"
id 2 1
"conference vldb 05 theobald"
title 3 2
"efficient versatile engine topx search"
abs 4 3
"present novel engine coin topx rank retrieve xml document"
sec 5 6
"introduction non schema xml data"
sec 8 9
"relate work efficient evaluation xml path condition"
st 6 4
"introduction"
par 7 5
"non schema xml data"
st 9 7
"relate work"
par 10 8
"efficient evaluation xml path condition"

Fig. 2. Redundant full-content text nodes for elements

2.2 Content Scores

TopX provides the option to evaluate queries either in conjunctive mode or in "andish" mode. In the first case, all terms and structural conditions must be met by a result candidate, but still different matches yield different scores. In the second case, a node matches a content condition of the form `//"`t_1 t_2`..."` if its content contains at least one occurrence of at least one of the terms t_1, t_2, etc. It matches the full-content condition `.//"`t_1 t_2 `..."` if its full-content contains at least one occurrence of at least one of the search terms. In the first case, the significance (e.g., derived from frequencies and element-specific corpus statistics) of a matched term influences the score and the final ranking, but – similarly to boolean XPath – documents (or subtrees) that do not contain a specified term at all or that do not strictly match all structural query conditions are dismissed.

For content scores we make use of element-specific statistics that view the content or full-content of each element node n with tag A as a bag of words:

1) the *term frequency*, $tf(t, n)$, of term t in node n, which is the number of occurrences of t in the content of n;
2) the *full-content term frequency*, $ftf(t, n)$, of term t in node n, which is the number of occurrences of t in the full-content of n;

3) the *tag frequency*, N_A, of tag A, which is the number of nodes with tag A in the entire corpus;
4) the *element frequency*, $ef_A(t)$, of term t with regard to tag A, which is the number of nodes with tag A that contain t in their full-contents in the entire corpus.

Now consider a content condition of the form `A//"`$t_1 \ldots t_m$`"`, where `A` is a tag name and t_1 through t_m are terms that should occur in the full-contents of a subtree. Our scoring of node n with regard to condition `A//"`$t_1 \ \ldots t_m$`"` uses formulas of the following type:

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

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

An important lesson from text IR is that the influence of the term frequency and element frequency values should be sublinearly dampened to avoid a bias for short elements with a high term frequency of a few rare terms. Likewise, the instantiation of compactness in the above formula should also use a dampened form of element size. Highly skewed score distributions would be beneficial for candidate pruning (and fast algorithm termination), but typically at a high expense in retrieval quality. To address these considerations, we have adopted the popular and empirically usually much superior Okapi BM25 scoring model (originating in probabilistic IR for text documents [6]) to our XML setting, leading to the following scoring function:

Tag	N	avg(length)	k_1	b
`article`	16,808	2,903	10.5	0.75
`sec`	96,481	413	10.5	0.75
`p`	1,022,679	32	10.5	0.75
`fig`	109,230	13	10.5	0.75

Table 1. Element-specific parameterization of the extended BM25 model

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

with

$$K = k_1 \left((1 - b) + b \frac{length(n)}{avg\{length(n') \mid n' \ with \ tag \ A\}}\right).$$

The BM25 formula provides a dampened influence of the ftf and ef parts, as well as a compactness normalization that takes the average compactness of each element type into account. A simple hill-climbing-style parameter optimization using the 2004 INEX collection and relevance assessments yields a maximum in the MAP value for k_1 being set to 10.5, whereas the b parameter is confirmed to perform best at the default value of 0.75 provided in the literature. With regard to individual (element-specific) retrieval robustness, the above formula would also allow for a more elaborated parameter optimization for individual element types which was not considered for the current setup.

2.3 Structural Scores

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

In non-conjunctive (aka. “andish”) retrieval, a result document (or subtree) should still satisfy most structural constraints, but we may tolerate that some tag names or path conditions are not matched. This is useful when queries are posed without much information about the possible and typical tags and paths or for vague content and structure (VCAS) search, where the structural constraints merely provide a hint on how the actual text contents should be connected. Our scoring model essentially counts the number of structural conditions (or connected tags) that are still to be satisfied by a result candidate d and assigns a small and constant score mass c for every condition that is matched. This structural score mass is combined with the content scores and aggregated with each candidate's $[worstscore(d), bestscore(d)]$ interval. In our setup we have set $c = 1$, whereas content scores were normalized to $[0, 1]$, i.e., we emphasize the structural query conditions. Note that it is still important to identify non-satisfiable structural conditions as early and efficiently as possible, because this can reduce the $bestscore(d)$ of a result candidate and make it eligible for pruning.

The overall score of a document or subtree for a content-and-structure (CAS) query is the sum of its content and structural scores. For content-only (CO) queries, i.e., mere keyword queries, the document score is the sum, over all terms, of the maximum per-term element scores within the same target element.

If TopX is configured to return entire documents as query results (e.g., for the CO/S-Fetch&Browse task), the score of a document is the maximal score of any subgraph matching a target element in the document; if otherwise the result granularity is set to elements, we may obtain multiple results according to the differently scored target elements in a document. The internal TopX query processor completely abstracts from the original query syntax (NEXI or XPath) and uses a full-fletched graph traversal to evaluate arbitrary query DAGs. Furthermore, the top-k-style nature of the engine does not require candidates to be fully evaluated at all query conditions, but merely relies on [$worstscore(d)$, $bestscore(d)$] bounds to determine the current top-k results and the *min-k* threshold for algorithm termination.

3 Database Schema and Indexing

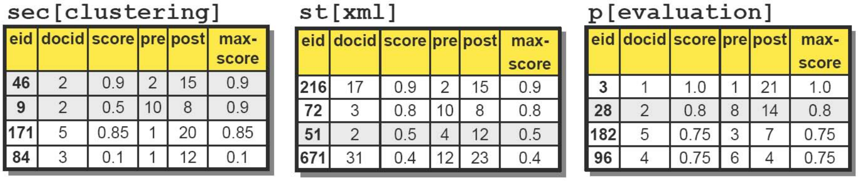
3.1 Schema

Inverted index lists are stored as database tables; Figure 3 shows the corresponding schema definitions with some example data for three tag-term pairs. The current implementation uses Oracle 10g as a backbone, mainly for easy maintenance of the required index structures, whereas the actual query processing takes place outside the database exclusively in the TopX query engine, such that the DBMS itself remains easily exchangeable. Nodes in XML documents are identified by the combination of document id (`did`) and preorder (`pre`). Navigation along all XPath axes is supported by both the `pre` and `post` attributes using the XPath accelerator technique of [3]. Additionally, the `level` information may stored to support the child-axis as well, but may be omitted for the NEXI-style descendant constraints. The actual index lists are processed by the top-k algorithm using two B$^+$-tree indexes that are created on this base table: one index for sorted access support in descending order of the (maxscore, did, score) attributes for each tag-term pair and another index for random access support using (did, tag, term) as key.

3.2 Inverted Block-Index

The base table contains the actual node contents indexed as one row per tag-term pair per document, together with their local scores (referring either to the simple content or the full-content scores) and their pre- and postorder numbers. For each tag-term pair, we also provide the *maximum score* among all the rows grouped by tag, term, and document id to extend the previous notion of single-line sorted accesses to a notion of *sorted block-scans*. Then TopX scans each list corresponding to the key (tag, term) in descending order of (maxscore, did, score). Each sequential block scan prefetches all tag-term pairs for the same document id in one shot and keeps them in memory for further processing which we refer to as *sorted block-scans*. Random accesses to content-related scores for a given document, tag, and term are performed through small range scans on the respective B$^+$ tree index using the triplet (did, tag, term) as key. Note that grouping tag-term pairs by their document ids keeps the range of the pre-/postorder-based in-memory structural joins small and efficient. All scores in the database tables are precomputed when the index tables are built.

For search conditions of the form `A[.//"`t_1 t_2`"]` using the descendants axis, we refer to the full-content scores, based on $ftf(t_1, A)$ and $ftf(t_2, A)$ values of entire document subtrees; these are read off the precomputed base tables in a single efficient sequential disk fetch for each document until the *min-k* threshold condition is reached and the algorithm terminates. We fully precompute and



sec[clustering]

eid	docid	score	pre	post	max-score
46	2	0.9	2	15	0.9
9	2	0.5	10	8	0.9
171	5	0.85	1	20	0.85
84	3	0.1	1	12	0.1

st[xml]

eid	docid	score	pre	post	max-score
216	17	0.9	2	15	0.9
72	3	0.8	10	8	0.8
51	2	0.5	4	12	0.5
671	31	0.4	12	23	0.4

p[evaluation]

eid	docid	score	pre	post	max-score
3	1	1.0	1	21	1.0
28	2	0.8	8	14	0.8
182	5	0.75	3	7	0.75
96	4	0.75	6	4	0.75

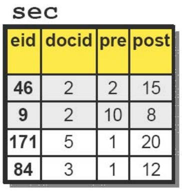
Fig. 3. Inverted block-index with precomputed full-content scores over tag-term pairs

materialize this inverted block index to efficiently support the NEXI-style descendant axis. With this specialized setup, parsing and indexing times for the INEX collection are about 80 minutes on an average server machine including the modified BM25 scoring model and the materialization of the inverted block-index view.

We propagate, for every term t that occurs in a node n with tag A, its local tf value "upwards" to all ancestors of n and compute the ftf values of these nodes for t. Obviously, this may create a redundancy factor that can be as high as the length of the path from n to the root. Thus, the redundant full-content indexing introduces a factor of redundancy for the textual contents that approximately corresponds to the average nesting depth of text nodes of documents in the corpus; it is our intention to trade off a moderate increase in inexpensive disk space (factor of 4-5 for INEX) for faster query response times. Note that by using tag-term pairs for the inverted index lookups, we immediately benefit from more selective, combined tag-term features and shorter index lists for the actual textual contents, whereas the hypothetical combinatorial bound of $\#tags \cdot \#terms$ rows is by far not reached.

3.3 Navigational Index

To efficiently process more complex queries, where not all content-related query conditions can be directly connected to a single preceding tag, we need an additional element-only directory to test the structural matches for tag sequences or branching path queries.



sec

eid	docid	pre	post
46	2	2	15
9	2	10	8
171	5	1	20
84	3	1	12

Fig. 4. Navigational index for branching path queries

Lookups to this additional, more compact and non-redundant navigational index yield the basis for the structural scores that a candidate may achieve for each matched tag-only condition in addition to the BM25-based content scores. As an illustration of the query processing, consider the example twig query `//A[.//B[.//"b"] and .//C[.//"c"]]`. A candidate that contains valid matches for the two extracted tag-term pairs `B:b` and `C:c` fetched through a series of block-scans on the inverted lists for `B:b` and `C:c`, may only obtain an additional static score mass c, if there is a common `A` ancestor that satisfies both the content-related conditions based on their already known pre-/postorder labels. Since all structural conditions are defined to yield this static score mass c, the navigational index is exclusively accessed through random lookups by an additional B$^+$ tree on this table. [10] provides different approaches to judiciously schedule these random accesses for the most promising candidates according to their already known content-related scores.

3.4 Random Access Scheduling

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

The first strategy, coined *MinProbe*, aims at a minimum number of random accesses by probing structural conditions for the most promising candidates, only. Since we do not perform any sorted scans for elementary tag conditions, we treat structural conditions as *expensive predicates* in the sense of [1]. We schedule random accesses only for those candidates d whose $worstscore(d)+o_j \cdot c > min\text{-}k$, where o_j is the number of untested structural query conditions for d and c is a static score mass that d earns with every satisfied structural condition.

This way, we schedule a whole batch of random lookups, if d has a sufficiently high $worstscore(d)$ to get promoted to the top-k when the structural conditions can be satisfied as well. If otherwise $bestscore(d)$ already drops below the current *min-k* threshold after a random lookup, we may safely prune the candidate from the queue. More sophisticated approaches may employ an analytic cost model, coined the *BenProbe* strategy in [10], in order to determine whether it is cost beneficial to explicitly lookup a candidate's remaining score in the structural and content-related query conditions.

4 Expensive Text Predicates

The use of auxiliary query hints in the form of expensive text predicates such as phrases (“”), mandatory terms (+), and negation (-) can significantly improve the retrieval results of an IR system. The challenge for a top-k based query processor lies in the *efficient* implementation of these additional query constraints and their adaptation into the sorted versus random access scheduling paradigm.

4.1 Negation

The semantics of negations in a non-conjunctive, i.e. “andish”, query processor is all but trivial. To cite the authors of the NEXI specification, “a user would be surprised if she encountered the negated term among the retrieval results”. This leaves some space for interpretation and most commonly leads to the conclusion that the negated term should not occur in any of the top-ranked results; yet we do not want to eliminate all elements containing one of the negated terms completely, if they also contain good matches to other content-related query conditions, and we would run into the danger of loosing substantial amount of recall. Therefore the scoring of negated terms is defined to be independent of the term's actual content score. Similarly to the structural query constraints introduced in the previous section, an element merely accumulates some additional static score mass if it does not match the negated term. This quickly leads us back to the notion of expensive predicates and the minimal probing approach. A random lookup onto this element's tag-term offsets is scheduled, if the element gets promoted into the top-k results after a “successful” negation test, i.e., if it does not contain the negated term among its full-content text and obtains the static score for the unmatched negation. In the current setup, this static score mass was set to the same value $c = 1$ that was provided for structural query constraints.

4.2 Mandatory Terms

In contrast to term negations, the scores for mandatory query terms should still reflect the relevance of the term for a given element, i.e., the precomputed BM25-based content scores. Yet a too strict boolean interpretation of the + operator would make us run into the danger of loosing recall at the lower ranks. We therefore introduce boosting factors and a slightly modified score aggregation of the form $score(n, A//"t_1 \ldots t_m") = \sum_{i=1}^{m} \beta_i + s(t_i, A)$, where $s(t_i, A)$ is the original content score, and β_i is set to 1 if the term is marked as mandatory (+) and 0 otherwise. Note that these β_i are constants at query evaluation time, and since the modified scores are taken into account for both the $worstscore(d)$ and $bestscore(d)$ bounds of all candidates, the boosting factors “naturally” enforce deeper sequential scans on the inverted index lists for the mandatory query conditions, typically until the final top-ranked results are discovered in those lists. Still weak matches for the remaining non-boosted query conditions may be compensated by a result candidate through high-scored matches in the mandatory query conditions.

4.3 Phrases and Phrase Negations

For phrase matching we store all term offsets in an auxiliary database table together with the pre-/postorder labels of each term's occurrence in a document. Again, phrases are interpreted as expensive predicates and tested by random accesses to the offset table using the minimal probing approach already described for the MinProbe scheduling. The only difference now is to determine whether a candidate element may aggregate the content-related score mass for the phrase-related conditions into it's overall $worstscore(d)$ that is then used to determine its position in the top-k results. In order to keep these score aggregations monotonous in the precomputed content scores, phrase lookups are treated as binary filters, only. Similarly to the single-term negations, phrase negations are defined to yield a static score mass c for each candidate element that does not contain the negated phrase. Single-term occurrences of the negated phrase terms are allowed, though, and do not contribute to the final element score unless they are also contained in the remaining part of the query.

5 Experimental Results for TopX

5.1 CO-Thorough

For the CO-Thorough task, TopX ranks at position 22 for the nxCG[10] metric using a strict quantization with a value of 0.0379 and at rank 30 of 55 submitted runs for ep/gr with a MAep value of 0.018. We used the modified BM25 scoring model described above and also expensive text predicates to leverage phrases, negations, and mandatory terms. The very modest rank in the sensitive CO-Thorough task attests that there is still some space left for optimizations in our scoring model for the CO queries, when there is no explicit target element specified by the query ("//*"). However, there was neither any restriction given on result overlap or result granularities, nor on the expected specificity or exhaustiveness of special element types such as sections or paragraphs, such that the engine was allowed to return any type of element (also list items or even whole articles) according to their aggregated content scores. An additional simple postprocessing step based on the element granularities and overlap removal would already be expected to achieve great performance gains here. However, for the old precision/recall metrics using inex_eval with the strict quantization (INEX '04), the TopX run ranks at a significantly better position of rank 3 with a MAP value of 0.0581, which actually corresponds to the particular metric and setup for which we had been tuning the system.

5.2 COS-Fetch & Browse

The situation somewhat improves for the COS-Fetch&Browse task. Here, TopX was configured to first rank the result documents according to their highest-ranked target element and then return all target elements within the same result document, but mistakenly with the same, document-wide score instead of the elements' own scores. This bug practically rendered the element-level results useless, while the article-level evaluation is relatively promising: The TopX run without expensive predicates like phrases etc. ranks at position 10 out of 25 with a MAep of 0.1455 in the ep-gr metric for the article ranking with strict quantization. the run that exploits phrases, mandatory and negative terms was slightly worse at rank 13 with a MAep of 0.1351 for the same setting. For both runs, structure was evaluated according to a strict interpretation of the query target

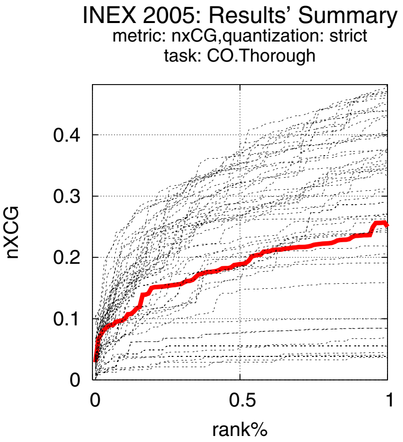


Fig. 5. nxCG results for the TopX CO-Thorough run

element given which exactly matches our full-content scoring model. Here, the strict – XPath-like – interpretation of the query target element in combination with our full-content scoring model that treats each target element itself as a mini-document shows its benefits and naturally avoids overlap, since we return exactly the element type that is specified in the query and therefore seem to match the result granularity expected by a human user in a much better way.

5.3 SSCAS

Finally, the SSCAS task perfectly matches our strict interpretation of the target element with the precomputed full-content scores and no overlap being allowed. The two submitted TopX runs rank at position 1 and 2 out of 25 submitted runs for both the strict nxCG[10] metric (with a value of 0.45 for both runs) and the strict ep/gr metric (with MAep values of 0.1334 for the run that considers expensive predicates and 0.1199 for the run without them). Although this strict evaluation might be less challenging from an IR point-of-view, this task offers most opportunities to improve the efficiency of a structure-aware retrieval system, because the strict notion of all structural query components like target and support elements drastically reduces the amount of result candidates per document and, hence, across the corpus. Clever precomputation of the main query building blocks, namely tag-term pairs with their full-content scores, and index structures for efficient sorted and random access on whole element blocks grouped by document ids allows for decent run times of a true graph-based query engine that lies in the order of efficient text IR systems. Here, TopX can greatly accelerate query run times and achieve interactive response times at a remarkable result quality. Similar experiments provided in [10] yield average response times for typical INEX (CO and CAS) queries in between 0.1 and 0.7 seconds for the top 10-20 and still an average run time of about 20 seconds for the top 1,500

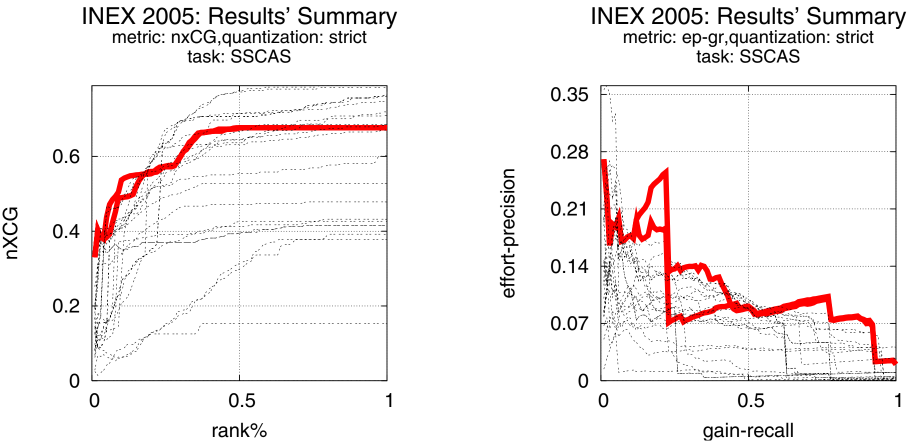


Fig. 6. TopX SSCAS runs

results as demanded by INEX (which is of course not exactly nice to handle for a top-k engine).

6 Experiments with XXL

The XXL Search Engine [7, 8, 9] was among the first XML search engines that supported content-and-structure queries with an IR-like scoring for content conditions. Focussing on aspects of semantic similarity conditions for tags and contents using ontologies, it applies an out-of-the-box text retrieval engine, namely Oracle's text engine, to evaluate content subqueries. Details of its architecture can be found in [8].

6.1 CO-Thorough

The CO.Thorough run basically represents the performance of the underlying text search engine. XXL automatically converted CO topics into corresponding Oracle text queries, using conjunctive combination of terms, enabling phrases, and applying some other simple heuristics that gave reasonable results with INEX 2003 and 2004. Surprisingly, this year's performance was not really convincing, with a rank 39 of 55 with inex_eval and the strict quantization (MAP 0.016), with similar results for the other metrics.

6.2 SSCAS

The results for the SSCAS run, where XXL has a higher influence on the outcome than with keyword-only topics, were much better. XXL is almost consistently among the top 10 for nxCG with the generalized quantization, with a peak rank of 2 for nxCG[25], and only slightly worse for strict quantization. For inex_eval, we achieved rank 11 with a MAP of 0.075. XXL has been especially built for this kind of strict structural match. The results are even better when taking the poor performance of the content-only run into account.

6.3 SVCAS and VVCAS

For the SSCAS run, XXL was configured to return a result only if it had a corresponding match (i.e., an element) for each subcondition of the query. For the SVCAS run, we relaxed this requirement and allowed results as soon as they had a match for the target subcondition, i.e., the subcondition whose result is returned as result of the query. This simple, 'andish'-like evaluation did surprisingly well, with top-15 ranks in most metrics.

For the VVCAS run, we additionally changed the tag of the target subcondition to the wildcard '*', accepting any element as result as long as it matches the associated content condition. However, this kind of relaxation turned out to be too coarse, so the results were quite poor with all metrics.

References

1. K.-C. Chang and S.-W. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD 2002*, pages 346–357, 2002.
2. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.
3. T. Grust. Accelerating XPath location steps. In *SIGMOD 2002*, pages 109–120, 2002.
4. U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB 2000*, pages 419–428, 2000.
5. S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE 1999*, pages 22–29, 1999.
6. S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR*, pages 232–241, 1994.
7. R. Schenkel, A. Theobald, and G. Weikum. XXL @ INEX 2003. In *INEX 2003 Workshop Proceedings*, pages 59–68, 2004.
8. R. Schenkel, A. Theobald, and G. Weikum. Semantic similarity search on semistructured data with the XXL search engine. *Information Retrieval*, 8(4):521–545, December 2005.
9. A. Theobald and G. Weikum. Adding Relevance to XML. In *WebDB 2000*, pages 105–124, 2000.
10. M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for TopX search. In *VLDB 2005*, pages 625–636, 2005.
11. M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB 2004*, pages 648–659, 2004.