Retrieval Quality vs. Effectiveness of Relevance-Oriented Search in XML Documents

Norbert Fuhr*
University of Duisburg-Essen

Norbert Gövert
University of Dortmund

Mohammad Abolhassani
University of Duisburg-Essen
Germany

Content-only queries in hierarchically structured documents should retrieve the most specific document nodes which are exhaustive to the information need. For this problem, we investigate two methods of augmentation, which both yield high retrieval quality. As retrieval effectiveness, we consider the ratio of retrieval quality and response time; thus, fast approximations to the 'correct' retrieval result may yield higher effectiveness. We present a classification scheme for algorithms addressing this issue, and adopt known algorithms from standard document retrieval for XML retrieval. As a new strategy, we propose *incremental-interruptible retrieval*, which allows for instant presentation of the top ranking documents. We develop a new algorithm implementing this strategy and evaluate the different methods with the INEX collection.

1 Introduction

With the increasing availability of documents in XML format, there is a growing need for retrieval methods exploiting the specific features of this type of documents. In 2002, INEX (initiative for the evaluation of XML retrieval) started, with the goal of defining standard retrieval tasks for XML documents and developing appropriate evaluation methods [Fuhr et al. 03]

Since XML documents contain explicit information about their logical structure, XML retrieval methods should take into account the structural properties of the documents to be retrieved. One of the two tasks considered in INEX deals with *content-only queries*, where only the requested content is specified. Instead of retrieving whole documents, the IR system should aim at selecting *document components* that fulfill the information need. Following the FERMI model [Chiaramella et al. 96], these components should be the deepest components in the document structure, i. e. most specific, while remaining exhaustive to the information need.

In this paper, we call this task also "relevance-oriented search". Like for any kind of retrieval, there are two major issues to be addressed:

Retrieval quality: How can we retrieve relevant documents, by avoiding nonrelevant ones? Whereas the FERMI model is easily applicable in the case of binary indexing (select the smallest component containing all query terms), the usage of weighted indexing requires a method balances between indexing weights and component specifity. For this purpose, we have developed the method of *augmentation*, which gave us very good results in the first INEX round.

Efficiency: How can we retrieve the top-ranking answers in very short time? Given the fact that not only complete documents, but (in principle) any XML element may be retrieved (and any element containing at least one of

*fuhr@uni-duisburg.de

the search terms is a candidate answer), we need appropriate methods for dealing with this extended search space.

In this paper, we take an integrated view on retrieval quality and efficiency, by focusing on *effectiveness*: Roughly speaking, the effectiveness of a retrieval method is the quality of the retrieval result, divided by the amount of user effort for achieving this result. Most IR test settings assume a constant user effort, thus it is sufficient to regard retrieval quality only. However, retrieval times for structured or multimedia documents may be substantial, thus the variance in user effort resulting from the response time cannot be neglected. In this setting, a high retrieval effectiveness can be achieved either by improving retrieval quality or by reducing retrieval time. In this paper, we will first describe a method giving us high retrieval quality, and then focus on the latter aspect.

Although users are interested only in the top n documents of the result ranking only, algorithms focusing on these elements only achieve very little efficiency gains over standard methods computing the full ranking (see e.g. [Buckley & Lewit 85]). Thus, we are looking for approximation methods, accepting small losses of retrieval quality, but expecting big gains in terms of efficiency. So the major theme of this paper is improvement of retrieval effectiveness by dealing with the tradeoff between retrieval quality and efficiency.

For this purpose, we take known solutions to this problem developed for retrieval of atomic (complete) documents, adopt them for XML retrieval and compare their performance. In addition, we propose a new strategy for dealing with quality vs. efficiency: In *interrupt-driven retrieval* the user does not have to wait for the system to provide an answer; instead, s/he may request immediate delivery of the best answer currently available. We present a new retrieval algorithm following this strategy, and propose an appropriate evaluation method. The experimental evaluation of the different approaches is based on the INEX test bed.

The remainder of this paper is structured as follows: In the next section, we briefly describe our augmentation method and present two variants that give comparable retrieval performance. The following section presents several algorithms addressing the tradeoff between retrieval quality and efficiency, and gives a classification scheme for these methods. Section 4 describes how these algorithms can be applied for retrieval of structured documents. Experimental results are presented in section 5, followed by the final conclusions.

2 Relevance-oriented search based on augmentation

The goal of relevance-oriented search is to retrieve the most specific document components that satisfy the query. For this purpose, two issues have to be addressed:

1. Irrespective of the query, what are possible retrieval answers? Given the fine-grained mark-up of many XML documents (down to the word level or even below), not any element may represent a meaningful answer.
2. How can we combine the criterion of specifity of an answer with weighted indexing? Here we need a weighting method for specifity of nodes, in order to be compute a combined weight.

For the first problem, we have adopted the notion of *index nodes* from the FERMI model: Given the XML document type definition (DTD), only certain elements are to be considered as possible retrieval answers. Thus, we assume that the database administrator has defined these element types beforehand.

For computing indexing weights, we want to apply standard methods developed for atomic documents. For this purpose, we have to split each document into disjoint parts. On the other hand, index nodes may be nested. Thus, we start with the most specific index nodes, treating their textual content like an atomic document. For the higher-level index nodes, only the text that is not contained within the other index objects is indexed. As an example, assume that we have defined section, chapter and book elements as index nodes. Then the title and introduction of a chapter would form this additional text chunk, since they do not belong to the sections included in the chapter.

When performing retrieval, the index weights of the most specific index nodes are given directly. For retrieval of the higher-level nodes, we have

to combine the weights of the different text units contained therein. For example, assume the following excerpt of a document structure, where we list the weighted terms instead of the original text:

```
<chapter> 0.3 XPath
  <section> 0.5 example </section>
  <section> 0.8 XPath 0.7 syntax </section>
</chapter>
```

A straightforward possibility would be the logical disjunction of the different weights for a single term. As an example, the Boolean query 'syntax AND example' can be answered by the complete chapter only, with a weight of $0.5 \cdot 0.7 = 0.35$ (assuming independence of terms). However, searching for the term "XPath" in this example would retrieve the whole chapter in the top rank ($0.3{+}0.8{-}0.3{\cdot}0.8 = 0.86$), whereas the second section would be given a lower weight (assuming that term occurrences in different nodes are also independent). Obviously, this strategy always assigns the highest weight to the most general element, thus contradicting the FERMI model mentioned before. As a better solution, we adopt the augmentation strategy from [Fuhr et al. 98], where index term weights have to be down-weighted when they are propagated upwards to the next index node.

The most obvious method for propagation is multiplication by a propagation weight, as proposed in [Fuhr & Großjohann 01]. As a probabilistic interpretation, we assume that for each term weight being propagated to the next higher index node, there is an additional probabilistic event, so that the indexing weight has to be multiplied by the propagation weight. Therefore, we call this method *conditional propagation*. In our example, using a propagation weight of 0.3, the section weight of the term "XPath" would yield $0.3 \cdot 0.8 = 0.24$ at the chapter level. Combining this propagated weight with the indexing weight of the chapter would yield an augmented weight of $0.3 + 0.24 - 0.3 \cdot 0.24 = 0.396$; thus a query with this term would rank the section ahead of the chapter.

As an alternate method of augmentation, we assume an exponential form, where we take the counter-probability to the power of a propagation weight γ. This method is called *potential propagation* here. When propagating a weight u of a term upwards to the next index node level, the resulting weight is computed as $1 - (1-u)^{\alpha}$. In our example document, the section weight 0.8 of the term "XPath" combined with a propagation weight of 0.2 would yield 0.383, thus resulting in a combined weight of $0.3{+}0.383{-}0.3{\cdot}0.383 = 0.568$.

In order to describe the augmentation process more formally, let us assume that a document consists of a set of index nodes $N = \{n_1, \ldots, n_q\}$. For describing the tree structure, we use a relation $\prec \subseteq N \times N$, where $n_j \prec n_k$ iff n_j is a ancestor of n_k. Furthermore, there is a function $\lambda : N \to \mathbb{N}_0$ giving the level of a node, where the root has level 0 and $\lambda(n_j) = \lambda(n_k) + 1$ iff n_j is a child of n_k.

Let $u_{im} = P(t_i|n_m)$ denote the probabilistic indexing weight of term t_i wrt. node n_m. Since we are dealing with logical disjunctions of independent probabilistic events (occurrences of the same term in different indexing nodes), we will mainly consider counter-probabilities below, where $\bar{u}_{im} = 1{-}u_{im}$. The augmented weight w_{im} of term t_i wrt. node n_m is then the combination of its own indexing weight and the propagated weights for t_i from its descendants. For this purpose, let us assume a propagation function $p(u_{ij}, n_j, n_m)$ which propagates the weight of term t_i in node n_j to node n_m (provided that $n_m \prec n_j$). Thus, the augmentation function $a(n_m, t_i)$ for computing the weight of node n_m for term t_i can be defined as

$$\bar{w}_{im} = a(n_m, t_i) = \bar{u}_{im} \prod_{n_m \prec n_j} p(u_{ij}, n_j, n_m) \quad (1)$$

(where $\bar{w}_{im} = 1 - w_{im}$). The general forms of the two propagation functions discussed for our example document above can be defined as follows:

$$p_1(u_{ij}, n_j, n_m) = 1 - u_{ij} \cdot \alpha_1(n_m, n_j) \quad (2)$$

$$p_2(u_{ij}, n_j, n_m) = \bar{u}_{im}^{\alpha_2(n_m, n_j)} \quad (3)$$

In case the propagation weight $\alpha_2(n_m, n_j)$ is additive, (i.e. $\alpha_2(n_m, n_j) = \alpha_2(n_m, n_k) + \alpha_2(n_k, n_j)$ iff $n_m \prec n_k \prec n_j$), we can derive a recursive form of the augmentation function, such that we must consider only the augmentation weights w_{ij} of the children n_j of node n_m, and not the indexing weights of all its ancestors:

$$a(n_m, t_i) = \bar{u}_{im} \prod_{n_m \prec n_j \wedge \lambda(n_j) = \lambda(n_m)+1} \bar{w}_{ij}^{\alpha_2(n_m, n_j)} \quad (4)$$

In principle, the propagation weight functions $\alpha_i(n_m, n_j)$ can be defined node(-pair)-specific, taking into account e.g. the number of children/descendants of n_m as well as the length of the path between n_m and n_j. Here we assume a constant factor (β for α_1 and γ for α_2 — making the latter additive) for propagating from one index node level to the next one. Thus, the propagation functions can be simplified to

$$p_1(u_{ij}, n_j, n_m) = 1 - u_{ij} \cdot \beta^{\lambda(n_j)-\lambda(n_m)} \quad (5)$$

$$p_2(u_{ij}, n_j, n_m) = \bar{u}_{im}^{\gamma\cdot(\lambda(n_j)-\lambda(n_m))} \quad (6)$$

In the remainder of this paper, we only consider these special forms of conditional (p_1) and potential (p_2) propagation.

3 Efficiency vs. retrieval quality

The major goal of this paper is the investigation of the tradeoff between efficiency and retrieval quality, in order to optimize effectiveness. Especially, we are looking at fast approximation methods, that yield approximations to the 'correct' (in terms of the underlying retrieval method) retrieval result, but require less computational effort. With regard to this problem, we can classify retrieval algorithms along two dimensions, namely processing mode and control mode:

Processing mode describes how the result of a query is produced:

- **block:** The complete result is computed as one block.
- **incremental:** Results are produced iteratively, such that top-ranking documents are determined first, and continuing operations produce the next elements from the ranked list.

Control mode: specifies how the production of results (and especially the tradeoff between efficiency and retrieval quality) can be controlled:

- **fixed:** There is no possibility to control this tradeoff; usually, the correct result is computed in this way.
- **tunable:** There is a tuning parameter which affects the quality of the approximation to the correct result; this parameter has to be set before processing starts.
- **interruptible:** At any point in time, the user may ask for the next result element, and the system outputs the best one currently available. Given enough time, the system would compute the correct result.

In principle, one can imagine algorithms for all possible combinations of values for these two dimensions; e.g., a block-interruptible algorithm would yield a complete ranked list at the moment the user requests it.

In the following, we will briefly describe some approaches (for retrieval of atomic documents) published in the literature and classify them according to this scheme. Since most approaches in information retrieval are based on linear retrieval functions, we will restrict our discussion to this type of function. Furthermore, like most algorithms, we will only allow for accesses to the inverted file (and thus ignore approaches involving direct access methods, for retrieving the weight of a specific document for a given term, see e.g. [Fagin 99]).

First, let us introduce some additional notations: a query q consists of a list of search terms $(t_1, \ldots, t_l)$, which are ordered by decreasing query term weights $c_1 \geq \ldots \geq c_l$. For a document d_m, let u_{im} denote the indexing weight for term t_i. Then the linear retrieval function is defined as

$$r(q, d_m) = \sum_{i=1}^{l} c_i \cdot u_{im}$$

Assuming an inverted file structure, there is an inverted list for each term t_i in the collection, The list consists of pairs (o_m, u_{im}), where o_m is the identifier (object id, document number) of document d_m and u_{im} gives the weight of term t_i for this document. In most cases, the entries are sorted by increasing document number, to allow for run-length compression and more efficient processing. However, we will also consider other orderings below.

The standard algorithm for computing the 'correct' result first creates an accumulator A_m for each document d_m in the collection and initializes it with a weight of 0. Then the inverted file entries are processed term-wise, adding the product $c_i \cdot u_{im}$ to accumulator A_m. Once all entries are processed, the accumulators containing the top k

retrieval status values (RSVs) are determined and the corresponding documents are retrieved. So the standard algorithm is a block-fixed algorithm according to our classification

[Moffat & Zobel 96] describes two algorithms for computing approximate retrieval results, which take as tuning parameter the number of documents (accumulators) to be considered. Here accumulators are created only when a new document number is encountered, and inverted lists are accessed in the order of increasing query term weights. After each inverted list, the algorithm checks if the specified number of accumulators has been reached. If this condition is fulfilled, there are two variants for proceeding further: The *quit algorithm* stops processing of inverted file entries and ranks the accumulators created so far. In contrast, the *continue algorithm* processes also the remaining inverted lists, but does not create any additional accumulators; thus, the correct RSVs are computed for those documents seen before, which, in turn, form the input to the ranking step. The continue algorithm is more efficient than than the standard algorithm due to a new data structure and algorithm which allows for skipping entries in the remaining inverted lists. A predecessor of the quit algorithm is described in [Buckley & Lewit 85], which uses a different set of tuning parameters: besides the number k of requested documents, also a value $m \leq k$ must be specified; the algorithm stops when it can guarantee that m of the top k documents have been computed correctly. All these algorithms are block-tunable according to our classification scheme.

In [Pfeifer & Pennekamp 97], a general class of algorithms for incremental computation of the ranked list is described. For each accumulator A_m, it also maintains the set of terms T_m from the query for which inverted file entries have been read for this document. During processing, upper and lower bounds for the RSV in each accumulator are computed, and accumulators are ordered by their upper bound. Whenever there is a document whose lower bound is greater or equal to the upper bound of the current top-ranking of the remaining documents, it can be written to the output list. So these algorithms can be characterized as incremental-fixed. By modifying the output criterion, also incremental-tunable algorithms are possible.

A specific instance of this class of algorithms is the Nosferatu algorithm, which assumes that inverted list entries are sorted by decreasing indexing weights (a similar algorithm has been described in [Persin et al. 96]). In this case, inverted lists are processed in parallel, and entries are read in the order of decreasing RSV increments.

Table 1 gives a survey over the algorithms discussed in this section.

In this paper, we will investigate an incremental-interruptible version of the Nosferatu algorithm which we call Nosferatu*. For this purpose, we simply sort accumulators according to their current values. Whenever a document is requested, we output the current top-ranking element and exclude it from further computations. So this algorithm consists of the following steps:

1. Start with an empty set of accumulators and an empty output list.
2. Read the inverted list entry (o_m, u_{im}) with the highest RSV increment $c_i \cdot u_{im}$.
3. Create an accumulator $A_m := 0$ and term set $T_m := \emptyset$ if this is the first occurrence of document d_m.
4. $A_m := A_m + c_i \cdot u_{im}$; place A_m at the correct position in the ranked list of accumulators.
5. If there are any unread inverted list entries, proceed with step 2; otherwise stop.
6. Whenever a document is requested, output the top-ranking one and delete it from the set of accumulators.

4 Retrieval algorithms for structured documents

In the previous section, we have discussed algorithms for atomic documents only. For relevance-oriented search in XML documents, we also have to consider the document structure, and store appropriate information in the inverted list entries. Since we do not consider structural conditions here, it is sufficient to provide information about the hierarchical structure only, without considering XML element types. For describing the hierarchical structure, it is sufficient to enumerate the nodes at each node level; in order to support augmentation during indexing time, we must specify

	fixed	tunable	interruptible
block	standard	quit, continue, Buckley/Lewit	
incremental	Nosferatu		Nosferatu*

Table 1: Classification of algorithms

the complete path from the document root to the current node, by giving the list of corresponding node indexes.

Assuming that all entries for a document are stored together, a group of entries for all occurrences of a term in a document has the structure $(o_m, f_m, (e_1, \ldots, e_{f_m}))$. Again, o_m denotes the document number; f_m gives the number of nodes in this document containing the current term. For each of these nodes, there is an entry $e_j = (l_j, (n_1, \ldots, n_{l_j}), u_{ij})$, where l_j specifies the length of the path $(n_1, \ldots, n_{l_j})$ to the current node, and finally u_{ij} gives the indexing weight for this node. The entries within a document are stored in inorder sequence. This way, augmentation can be performed easily at retrieval time. As an example, the following document entry lists 5 occurrences of terms, with path lengths between 1 to 4: `(1234, 5, (1,(1),0.5), (3,(1,1,2),0.3), (3,(1,1,4),0.2), (4,(1,2,7,3),0.4), (2,(2,3),0.6))`. A method for compressing this type of inverted lists is described in [Thom et al. 95]

This basic structure can be varied in a number of ways:

weights stored: We can store either indexing weights or augmentation weights. The latter is only possible for potential propagation, or in case we store weights for all nodes to which indexing weights are propagated. Storing only indexing weights allows for choosing the propagation method and weight at retrieval time, whereas storage of augmentation weights can speed up the retrieval process.

degree of augmentation: In case we store augmentation weights, we can choose for which nodes we create an occurrence entry: At least, we must consider all nodes which have been assigned indexing weights. The other extreme would be the storage of entries for all nodes to which indexing weights are propagated. Intermediate solutions are also possible, where different criteria can be used for choosing the nodes to be considered.

ordering of entries Usually, inverted file entries are stored in the order of increasing document numbers. For applying the Nosferatu* algorithm, we also test two other sorting orders. As additional data, we compute the maximum augmentation weight of a node in the document, and then sort document entries according to decreasing values of these weights. However, retrieval with this sorting order requires the consideration of all nodes with nonzero weights within a document where possibly only one node has a high weight. Therefore, we also consider an inverted file organization were we give up document-wise grouping of occurrence entries, and store each occurrence along with its document number. Then we sort occurrence entries of the complete collection according to decreasing weights; of course, we can combine this organization with both types of weights and all degrees of augmentation.

5 Experiments

5.1 Test setting

For our experiments, we used the INEX test collection [Fuhr et al. 03]. The document collection is made up of the full-texts, marked up in XML, of 12 107 articles of the IEEE Computer Society's publications from 12 magazines and 6 transactions, covering the period of 1995–2002, and totalling 494 megabytes in size. Although the collection is relatively small compared with TREC, it has a suitably complex XML structure (192 different content models in DTD) and contains scientific articles of varying length. On average an article contains 1 532 XML nodes, where the average depth of a node is 6.9. A more detailed summary can be found in [Fuhr et al. 02]. For our experiments, we defined four levels of index

nodes for this DTD, where the following XML elements formed the roots of index nodes: article, section, ss1, ss2 (the latter two elements denote two levels of subsections).

As queries, we used the 24 content-only queries from INEX for which relevance judgments are available. As query terms, we considered all single words from the topic title, the description and the keywords section.

For relevance assessments, INEX uses a two-dimensional, multi-valued scale for judging about relevance and coverage of an answer element. In our evaluations described below, recall and precision figures are based on a mapping of this scale onto a one-dimensional, binary scale where only the combination 'fully relevant'/'exact coverage' is treated as relevant and all other combinations as non-relevant. For each query, we considered the top ranking 1000 answer elements in evaluation.[1].

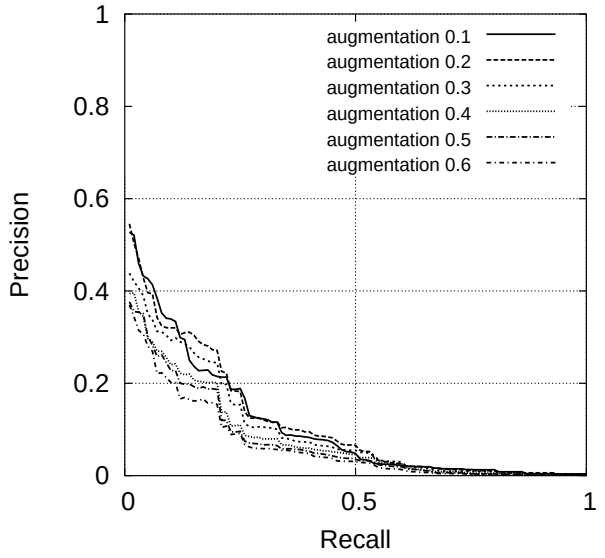

Figure 1: Impact of propagation weights on retrieval quality

5.2 Retrieval based on augmentation

In the first series of experiments, we evaluated the retrieval quality of our augmentation-based approach. For computing the indexing weights of terms in index nodes, we applied the standard BM25 formula [Robertson et al. 92]. As query term weights, *tf* weights were used.

When comparing the retrieval quality of conditional and potential augmentation, it turned out that both methods yield nearly identical results; they only differ in the absolute RSV values. However, the choice of the propagation weight has a substantial effect on performance, as can be seen in the recall-precision curves (Figure 1) as well as in the average precision for 100 recall points (Table 2). In INEX 2002, we submitted 2 official runs with conditional augmentation using weights of 0.3 and 0.6. The former gave the best performance among all official runs. From Table 2, we learn that we can even improve our performance by choosing a propagation weight of 0.2. Thus, all experiments described below use this weighting factor. Furthermore, since there are no differences between conditional and potential augmentation, we only consider potential augmentation in the following.

propagation weight	average precision
0.1	10.70 %
0.2	11.21 %
0.3	9.78 %
0.4	8.01 %
0.5	7.32 %
0.6	6.43 %

Table 2: Average precision for retrieval with different propagation weights

5.3 Predictors of retrieval time

In order to investigate retrieval effectiveness, we must measure retrieval time. However, exact timings depend on a large variety of factors related to both hardware (e.g. CPU, main memory, disk) and software parameters (e.g. implementation language, operating system). Therefore, we think that relative timings are sufficient for the issues investigated in this paper. Instead of measuring retrieval times directly (with all the related problems of stochastic experiments and comparability of implementations), we were looking for other parameters that are good predictors of retrieval time. For this purpose, we measured retrieval times for the 30 INEX CO queries and

[1] The official INEX 2002 evaluation was based on the top 100 elements only — for details of the INEX evaluation, see [Gövert & Kazai 03].

plotted the actual timings against two parameters:

1. sum of all document frequencies of query terms,
2. sum of all index node frequencies of query terms.

Moreover, we ran these experiments with two different indexing schemes: Besides the standard scheme with four levels of index nodes as described above, we considered a scheme with six levels, where also body and paragraph elements were treated as index node roots[2]. The scatterplots shown in Figure 2 and the respective correlation coefficients displayed in Table 3 indicate a strong linear relationship between retrieval times and both types of term frequencies. Thus, any of these frequencies is a good predictor of retrieval time. In the following, index node frequencies are used for this purpose.

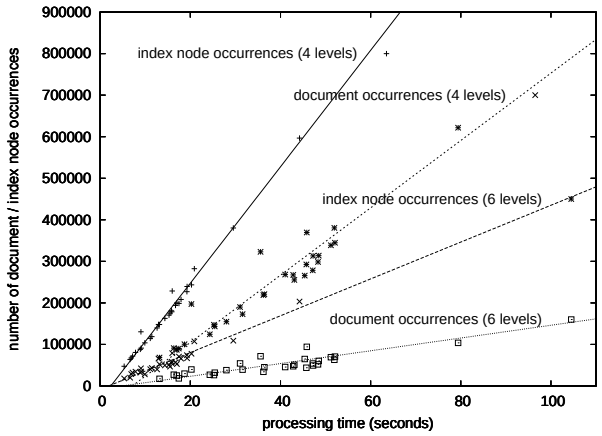

Figure 2: Retrieval time vs. number of document / index node frequencies

index node frequencies	4 levels	0.99
document frequencies	4 levels	0.96
index node frequencies	6 levels	0.98
document frequencies	6 levels	0.92

Table 3: Correlation coefficients for linear dependency of document / index node frequencies and retrieval time

[2]Retrieval quality with six levels of index nodes was worse than with four levels.

5.4 Quit and continue algorithm

We implemented a variant of the quit and continue algorithms mentioned in Section 3. Instead of specifying the number of accumulators to be used, our tuning parameter is the percentage of query terms to be considered; here query terms are ordered by their $tf \cdot idf$ values. Table 4 gives the corresponding figures.

For the quit algorithm, efficiency depends linearly on the number of document postings read from the inverted lists; thus, we have a nice tradeoff between efficiency and retrieval quality, which provides substantial savings in terms of retrieval time in combination with small performance degradation.

For the continue algorithm, we implemented two versions: In the continue/document variant, skipping was performed at the document level only; for a document to be considered, all occurrence entries were taken into account; thus weights in accumulators could also be affected through propagated weights. In contrast, the continue/inode variant only considered occurrence entries that contained indexing weights for nodes for which there was an accumulator, and weight propagation to other nodes was suppressed.

The performance figures for both variants show that the continue strategy is much better than the quit method. Among the two continue variants, retrieval quality of continue/inode is not reduced for up to 92% of postings ignored, with even substantial gains when only 60–80% of the query terms are considered in the first phase of the algorithm[3]. For explaining the difference in terms of retrieval quality between the two variants, we must keep in mind that continue/document is the correct one, whereas continue/inode does not perform propagation during the continue phase. So it seems that this 'refinement' of the propagation strategy leads to even better retrieval results.

Since we are only considering the number of postings to be skipped, we cannot guarantee that these gains are proportional to the reduction of retrieval time. The skipping algorithm for documents described in [Moffat & Zobel 96] has been tested for a collection where the number of docu-

[3]For the case of atomic documents, [Moffat & Zobel 96] also reports improvements in terms of retrieval quality when the number of accumulators is restricted.

term ratio	quit postings ignored	quit average precision	continue / document postings ignored	continue / document average precision	continue / inode postings ignored	continue / inode average precision
100 %	0 %	11.21 %	0 %	11.21 %	0 %	11.21 %
90 %	37 %	11.25 %	34 %	11.25 %	36 %	11.40 %
80 %	53 %	11.23 %	49 %	11.28 %	51 %	11.55 %
70 %	64 %	10.95 %	59 %	11.29 %	63 %	11.58 %
60 %	73 %	10.42 %	68 %	11.38 %	71 %	11.59 %
50 %	79 %	9.36 %	74 %	11.31 %	77 %	11.25 %
40 %	87 %	9.00 %	82 %	10.87 %	85 %	11.15 %
30 %	91 %	9.63 %	86 %	10.91 %	89 %	11.71 %
20 %	94 %	9.00 %	90 %	10.48 %	92 %	11.22 %
10 %	99 %	5.60 %	96 %	8.08 %	98 %	8.17 %

Table 4: Results for quit and continue algorithm

ments was two orders of magnitudes higher, but with unstructured documents. Thus, only experimentation with an appropriate implementation can solve this issue.

5.5 Nosferatu*

As described in Section 4, the Nosferatu* algorithm can be applied for different inverted file organizations. In our experiments, we tested the following variants:

inode/aweights Inverted list entries contain augmentation weights (full augmentation) and are ordered by decreasing values of these weights (no document-wise grouping of occurrence entries).

document/iweights Inverted list entries contain indexing weights (without augmentation), and entries are grouped document-wise. The document entries are sorted by maximum index node indexing weight.

document/aweights Like before, but instead of indexing weights, augmentation weights are stored and used as sorting criterion.

The inode/aweights variant is a kind of best case for the Nosferatu* algorithm, but has high requirements in terms of storage space needed for the inverted lists (a less expensive solution could e.g. be based on partial augmentation).

For comparison, we considered the **standard** algorithm, which gives us the upper bound of the performance, but the fixed-block mode requires that the user cannot view any document before retrieval is completed. As **baseline**, we regard an incremental-interruptible version of the standard algorithm: Here query terms are processed in the order of decreasing weights $tf \cdot idf$. Whenever an interrupt occurs, the current top-ranking document is submitted to the user.

As Nosferatu* is an incremental-interruptible algorithm, we need an appropriate metrics for evaluating this type of approach. Since there is no standard measure for effectiveness, we assume the following setting here: after submitting the query, the user issues interrupts at a constant rate in order to view the next document (as an alternative, one could also assume that s/he waits for some time before starting the first interrupt, but our results show that this is not necessary). For the interrupt intervals, we assume a rate of 1.5 interrupts per second, which is approximately the time a fast reader would need for skimming the title of a retrieved document.

As evaluation measure, we consider precision after n documents retrieved this way; in addition, we also consider average precision (for 100 recall points) as a measure of overall performance.

The retrieval results for the different methods are listed in Table 5. The most surprising result is the poor performance of the inode/aweights method, which is even below the baseline run. In contrast, the two Nosferatu* variants using document-wise grouping (document/iweights and document/aweight) perform quite well. With a Wilcoxon signed rank test, the document/* methods are significantly better (at the 95 % level)

	avg.	5	10	15	20
standard	11.21 %	29.57 %	26.96 %	22.03 %	20.00 %
document/aweights	10.30 %	27.83 %	25.65 %	21.45 %	19.57 %
document/iweights	9.37 %	29.57 %	25.65 %	21.45 %	18.91 %
inode/aweights	6.98 %	21.74 %	16.96 %	15.65 %	14.35 %
baseline	7.80 %	26.96 %	22.61 %	18.55 %	16.96 %

Table 5: Precision values for the Nosferatu* variants

than both the the baseline and the inode/aweights method.

One reason for the poor result of the inode/aweight variant is the fact that this approach partly ignores the document context: Since our queries contain 13.7 terms on average, we should retrieve index nodes that contain several of these terms. Without document-wise grouping of inverted list entries, our chances of locating these index nodes are rather low. In contrast, when using document-wise grouping, we get co-occurrences even when the high weights occur in different index nodes of the same document. This hypothesis is confirmed by the statistics about the average number of term hits per retrieved index node shown in Table 6. Using a Wilcoxon signed rank test, in 7 of the 8 cases the number of term hits for the document/* methods is significantly higher (at the 95 % level) than for the inode/aweights method.

# documents retrieved	5	10	15	20
document/aweights	4.01	4.49	4.51	4.42
document/iweights	3.67	4.23	4.31	4.24
inode/aweights	3.48	3.82	3.85	3.95

Table 6: Average number of term hits per index node retrieved

When comparing the two variants with document-wise grouping with the standard method, the performance loss is about 5% for the first 5–20 documents. The difference is significant only for document/iweight at 10 documents and document/aweight at 10 and 15 documents.

Given an average response time of about 30 seconds for the standard method in our current implementation, the tested interrupt rate of 1.5 per second means that the user has seen already the top 45 documents with Nosferatu* at the moment the standard method comes up with the blocked result. Especially for iterative retrieval (e.g. in combination with relevance feedback), Nosferatu* is clearly more effective.

The two document-oriented methods have similar performance: document/iweights performs better in the beginning (significant difference at 5, 10 and 20 documents), whereas document/aweights seems to be superior when more time is available.

Overall, we can conclude that the Nosferatu* algorithm in combination with document-wise grouping of inverted file entries fulfills our requirements for an incremental-interruptible algorithm. With the interrupt rate tested, it yields instant responses when a query is submitted, thus supporting highly interactive retrieval. The efficiency gains of Nosferatu* clearly outweigh its retrieval quality losses. Thus, this approach yields significant gains in terms of retrieval effectiveness.

6 Conclusions

In this paper, we have presented effective methods for relevance-oriented search in XML retrieval. For our augmentation-based approach, the experimental evaluation of two variants gave us a very good retrieval quality. Currently, we are developing methods for learning node-specific propagation weights, for which potential propagation is more suitable.

Instead of regarding only retrieval quality, we have emphasized an effectiveness-oriented view, where also efficiency must be taken into account. We first have developed a classification scheme for approaches in this area, and then investigated two types of algorithms in more detail. For two classical block-tunable algorithms, we have shown that they also can be applied successfully for XML retrieval, giving even better results than for unstruc-

tured documents. As a new class of approaches, we have proposed incremental-interruptible methods. As an instance of these methods, we have developed the Nosferatu* algorithm, which allows for instant retrieval of the top ranking documents, with only marginal losses of retrieval quality. So this algorithm yields very high effectiveness.

We think that algorithms of the latter kind are very important for interactive retrieval. Whereas holistic evaluation approaches are necessary for judging about the overall quality (see e.g. [Beaulieu & Robertson 96]), system-oriented approaches as considered here are necessary for tuning of components.

Overall, XML retrieval is a challenging new problem, for which appropriate retrieval methods are still at their infancy. With the major concepts presented in this paper — augmentation, effectiveness vs. quality, taxonomy of algorithms — we have addressed major issues for further research in this area.

References

Beaulieu, M.; Robertson, S. (1996). Evaluating Interactive Systems in TREC. *Journal of the American Society for Information Science 47(1)*, pages 85–94.

Buckley, C.; Lewit, A. (1985). Optimization of Inverted Vector Searches. In: *Proceedings of the 8th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 97–105. ACM, New York.

Chiaramella, Y.; Mulhem, P.; Fourel, F. (1996). *A Model for Multimedia Information Retrieval.* Technical report, FERMI ESPRIT BRA 8134, University of Glasgow.

Fagin, R. (1999). Combining Fuzzy Information from Multiple Systems. *Journal of Computer and System Sciences 58(1)*, pages 83–99.

Fuhr, N.; Großjohann, K. (2001). XIRQL: A Query Language for Information Retrieval in XML Documents. In: Croft, W.; Harper, D.; Kraft, D.; Zobel, J. (eds.): *Proceedings of the 24th Annual International Conference on Research and development in Information Retrieval*, pages 172–180. ACM, New York.

Fuhr, N.; Gövert, N.; Rölleke, T. (1998). DOLORES: A System for Logic-Based Retrieval of Multimedia Objects. In: Croft, W. B.; Moffat, A.; van Rijsbergen, C. J.; Wilkinson, R.; Zobel, J. (eds.): *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 257–265. ACM, New York.

Fuhr, N.; Gövert, N.; Kazai, G.; Lalmas, M. (2002). INEX: INitiative for the Evaluation of XML Retrieval. In: Baeza-Yates, R.; Fuhr, N.; Maarek, Y. S. (eds.): *Proceedings of the SIGIR 2002 Workshop on XML and Information Retrieval.*

Fuhr, N.; Gövert, N.; Kazai, G.; Lalmas, M. (eds.) (2003). *INitiative for the Evaluation of XML Retrieval (INEX). Proceedings of the First INEX Workshop. Dagstuhl, Germany, December 8–11, 2002*, ERCIM Workshop Proceedings, Sophia Antipolis, France. ERCIM. `http://qmir.dcs.qmul.ac.uk/inex/Workshop.html`.

Gövert, N.; Kazai, G. (2003). Overview of the INitiative for the Evaluation of XML retrieval (INEX) 2002. In [Fuhr et al. 03], pages 1–17. `http://qmir.dcs.qmul.ac.uk/inex/Workshop.html`.

Moffat, A.; Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems 14(4)*, pages 349–379.

Persin, M.; Zobel, J.; Sacks-Davis, R. (1996). Filtered Document Retrieval with Frequency-Sorted Indexes. *Journal of the American Society for Information Science 47(10)*, pages 749–764.

Pfeifer, U.; Pennekamp, S. (1997). Incremental Processing of Vague Queries in Interactive Retrieval Systems. In: Fuhr, N.; Dittrich, G.; Tochtermann, K. (eds.): *Hypertext – Information Retrieval – Multimedia (HIM).* Universitätsverlag Konstanz. `http://ls1-www.cs.uni-dortmund.de/HIM97/`.

Robertson, S. E.; Walker, S.; Hancock-Beaulieu, M.; Gull, A.; Lau, M. (1992). Okapi at TREC. In: *Text REtrieval Conference*, pages 21–30.

Thom, J. A.; Zobel, J.; Grima, B. (1995). *Design of indexes for structured document databases.* Technical Report TR-95-8, Collaborative Information Technology Research Institute, Melbourne, Australia.