# XPath Inverted File for Information Retrieval

Shlomo Geva
Centre for Information Technology Innovation
Faculty of Information Technology
Queensland University of Technology
GPO Box 2434 Brisbane Q 4001 Australia
s.geva@qut.edu.au

Murray Leo-Spork
Centre for Information Technology Innovation
Faculty of Information Technology
Queensland University of Technology
GPO Box 2434 Brisbane Q 4001 Australia
m.spork@qut.edu.au

## ABSTRACT

In this paper we describe the implementation of a search engine for XML document collections. The system is keyword based and is built upon an XML inverted file system. We describe the approach that was adopted to meet the requirements of Strict Content and Structure queries (SCAS) and Vague Content and Structure queries (VCAS) in INEX 2003.

**Keywords:** Information Retrieval, Inverted File, XML, XPath, INEX, Assessment, Evaluation, Search Engine

## 1. Introduction

Recently, the widespread use of Extensible Markup Language (XML) has led to appropriate Information Retrieval methods for XML documents [4]. A key difference between XML documents and conventional text documents is the separation of structure and content [5]. A standard solution for efficient Information Retrieval is to use an inverted file index. Zobel [6] identifies two dominate methods for indexing of large text databases: inverted files and signature files. Zobel compared these two methods and concluded that inverted files are superior in almost every respect, including speed, space and functionality.

In an inverted file, for each term in the collection of documents, a list of occurrences is maintained. Information about each occurrence of a term includes the document-id and term position within the document. Maintaining a term position in the inverted lists allows for proximity searches, the identification of phrases, and other context-sensitive search operators. This simple structure, combined with basic operations such as set-union and set-intersect, support the implementation of rather powerful keyword based search engines.

XML documents contain rich information about document structure. The objective of the XML Information Retrieval System that we describe in this paper is to facilitate access to information that is based on both content and structural constraints. We extend the Inverted File scheme in a natural manner, to store XML context in the inverted lists.

## 2. XML File Inversion

In our scheme each term in an XML document is identified by 3 elements. File path, absolute XPath context, and term position within the XPath context.

The file path identifies documents in the collection; for instance:

**C:/INEX/ex/2001/x0321.xml**

The absolute XPath expression identifies a leaf XML element within the document, relative to the file's root element:

**/article[1]/bdy[1]/sec[5]/p[3]**

Finally, term position identifies the ordinal position of the term within the XPath context.

One additional modification that we adopted allowed us to support queries on XML tag attributes. This is not a strictly content search feature, but rather structure oriented search feature. For instance, it allows us to query on the 2nd named author of an article by imposing the additional query constraint of looking for that qualification in the attribute element of the XML author element. The representation of attribute values is similar to normal text with a minor modification to the XPath context representation – the attribute name is appended to the absolute XPath expression. For instance:

**article[1]/bdy[1]/sec[6]/p[6]/ref[1]/@rid[1]**

Here the character '@' is used to flag the fact that "rid" is not an XML tag, but rather an attribute of the preceding tag <ref>.

An inverted list for a given term, omitting the File path and the Term position, may look something like this:

| Context |
|---|
| XPath |
| article[1]/bdy[1]/sec[6]/p[6]/ref[1] |
| article[1]/bdy[1]/sec[6]/p[6]/ref[1]/@rid[1] |
| article[1]/bdy[1]/sec[6]/p[6]/ref[1]/@type[1] |
| article[1]/bm[1]/bib[1]/bibl[1]/bb[13]/pp[1] |

| Context |
|---|
| XPath |
| article[1]/bm[1]/bib[1]/bibl[1]/bb[14]/pdt[1]/day[1] |
| article[1]/bm[1]/bib[1]/bibl[1]/bb[14]/pp[1] |
| article[1]/bm[1]/bib[1]/bibl[1]/bb[15] |
| article[1]/bm[1]/bib[1]/bibl[1]/bb[15]/@id[1] |
| article[1]/bm[1]/bib[1]/bibl[1]/bb[15]/ti[1] |
| article[1]/bm[1]/bib[1]/bibl[1]/bb[15]/obi[1] |

In principle at least, a single table can hold the entire cross reference list (our inverted file). Suitable indexing of terms can support fast retrieval of term inverted lists. However, it is evident that there is extreme redundancy in the specification of partial absolute XPath expressions (substrings). There is also extreme redundancy in full absolute XPath expressions where multiple terms in the same document share the same leaf context (e.g. all terms in a paragraph). Furthermore, many XPath leaf contexts exist in almost every document (e.g. /article[1]/fm[1]/abs[1]).

We have chosen to work with certain imposed constraints. Specifically, we aimed at implementing the system on a PC and base it on the Microsoft Access database engine. This is a widely available off-the-shelf system and would allow the system to be used on virtually any PC running under any variant of the standard Microsoft Windows operating system. This choice implied a strict constraint on the size of the database – the total size of an Access database is limited to 2Gbyte. This constraint implied that a flat list structure was infeasible and we had to normalise the inverted list table to reduce redundancy.

## 3. Normalized Database Structure

The structure of the database used to store the inverted lists is depicted in Figure 1. It consists of 4 tables. The ***Terms*** table is the starting point of a query on a given term. Two columns in this table are indexed - The ***Term*** column and the ***Term_Stem*** column. The ***Term_Stem*** column holds the Porter stem of the original term. The ***List_Position*** is a foreign key from the ***Terms*** table into the ***List*** Table. It identifies the starting position in the inverted list for the corresponding term. The ***List_Length*** is the number of list entries corresponding to that term. The ***List*** table is (transparently) sorted by Term so that the inverted list for any given term is contiguous. As an aside, the maintenance of a sorted list in a dynamic database poses some problems, but these are not as serious as might seem at first, and although we have solved the problem it is outside the scope of this paper and is not discussed any further. A search proceeds as follows. Given a search term we obtain a starting position within the List table. We then retrieve the specified number of entries by reading sequentially.

The ***Document*** and ***Context*** tables contain the actual file path and absolute XPath of a given term, respeciively. The inverted list for a given term is thus obtained by a *Join* (SQL) of the selected List table entries (as described above) with the ***Document*** and ***Context*** tables to obtain the complete de-normalised inverted list for the term. The XPath context is then checked with a regular expression parser to ensure that it satisfies the topic's <Title> XPath constraints.

The retrieval by ***Term_Stem*** is similar. First we obtain the Porter stem of the search term.

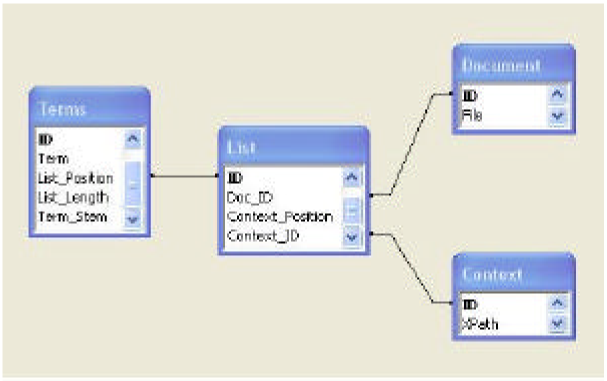

Figure 1: Database Schema for XML Inverted File.

Then we search the list by ***Term_Stem*** – usually getting duplicate matches. All the lists for the duplicate hits on the ***Terms*** table are then concatenated. The ***Context_Position*** is the ordinal position of the term within the leaf node of the article's XML tree. Phrases and other proximity constraints can be easily evaluated by using the ***Context_Position*** of individual terms in the ***List*** table.

We have not compressed XPath expressions to minimise the extreme redundancy of XPath substrings in the ***Context*** table. With this normalization the database size was reduced to 1.6GByte and within the Microsoft Access limits.

## 4. The CASQuery Engine

Before discussing the implementation details of the CASQuery engine it is necessary to introduce some terminology. We then describe the implementation of the search engine.

### 4.1 Terminology

- **XPath Query**: An XPath Query is a query that meets the criteria of the INEX query specification. It can be considered a subset of the W3C's XPath language.

- **Step**: A Step is a component of an XPath query that specifies some Axis (child, descendant, descendant-or-self etc.) a NodeTest (e.g. a NameText that tests the name of an element) and optionally some Predicate
- **Path**: A Path is a sequential list of Steps
- **Predicate**: A predicate contains a filter that specifies some condition that a node must meet in-order to satisfy it. This filter may be an "about" function or an equality expression.
- **Context**: The context for an element is an absolute XPath expression denoted by a list of child steps with a numerical index e.g. "/article[1]/bdy[1]/sec[1]/p[4]"
- **ReturnElement**: A ReturnElement is an element (qualified by the document name and a context) that satisfies the full path expression of a query (or query fragment) not including any path expression in a filter. The context of the ReturnElement is the one returned by the query engine to the user.
- **SupportElement**: A SupportElement is an element (qualified by the document name and a context) that satisfies the full path expression of a query (or query fragment) including any path expression in a filter. The context of the ReturnElement is not returned to the user but can be used to "support" the validity of the ReturnElement (in other words: shows why the ReturnElement was in fact returned).

The search engine was designed to operate on the <Title> element of CAS topics. It operates in the same manner for both strict (SCAS) and vague (VCAS) interpretation of the queries. The only difference is in the definition of equivalence tags:

SCAS Equivalent tags:

- **Article,bdy**
- **p|p[1-3]|ip[1-5]|ilrj|item-none**
- **sec|ss[1-3]**
- **h|h[1-2]a?|h[3-4]**
- **l[1-9a-e]|dl|list|numeric-list|numeric-rbrace|bullet-list**

VCAS Equivalent tags:

- **Article,bdy,fm**
- **sec|ss[1-3]|p|p[1-3]|ip[1-5]|ilrj| item-none**
- **h|h[1-2]a?|h[3-4]**
- **yr|pdt**
- **snm|fnm|au**
- **bm|bibl|bib|bb**
- **l[1-9a-e]|dl|list|numeric-list|numeric-rbrace|bullet-list**

## 4.2 Parsing the Query

We used the Programmar[2] parser development toolkit to generate a parser for XPath[3] queries. Programmar accepts a Backus Naur Form (BNF) grammar as input and is able to generate a parser that can parse an instance of that query into a *parse tree*. The Programmar library then provides an API to access and walk the parse tree that it constructed.

We used the XPath BNF grammar as defined by the W3C as input to the Programmar IDE. Some small modification to the BNF syntax was made in order to make the task of walking the parse tree and gathering the required information simpler.

Our approach was to walk the parse tree and construct an abstract syntax tree, which represents that same query but at a higher level of abstraction than the parse tree generated by the Programmer toolkit. Representing the query at a higher level of abstraction meant that implementing the query engine that processes that query was made simpler.

## 4.3 The Abstract Syntax

The abstract syntax was contained within a separate module that is kept independent of the QueryEngine that processes it. Thus we allow for the possibility that the abstract syntax for XPath queries may be utilised in other applications. For example it would be possible to implement a more traditional XPath processor on top of this abstract syntax. Therefore there is a dependency from the Query Engine to the Abstract Syntax package but no reverse dependency.

The basic structure of an XPath query (in the abstract syntax) is that it consists of a Path that contains a list of Steps. This is consistent with the terminology used by the XPath standard. Steps must contain a node test – and may also contain zero to many filters (or predicates).

## 4.4 Evaluateable Fragments

Once the XPath parser has constructed the abstract syntax, the query engine performs one further transformation on the query before executing. The path, or list of steps, must be broken down into EvaluateablePathFragments. Each step in the query that contains an *EvaluatableExpression* will be treated as the last step in an EvaluateablePathFragment.

An *EvaluatableExpression* is a step filter that can be evaluated by the QueryEngine.

In our implementation we are using an index of inverted lists that map a *term* to a list of *contexts* (full absolute XPath path plus document name). Therefore, for a filter to be *evaluateable* it must filter based on some term that can be looked up in the index. For example the filter:

**/article/bdy[count()=1]**

would not be *evaluateable* in our system as no terms is given in the filter. However the filter:

**/article//yr[. = "1999"]**

is evaluateable as the term "1999" will be in the index.

As an example, the query:

**//article[//yr='1999']//sec[about(./,'DEHOMAG')]**

would be broken down into two fragments:

1. **//article[//yr = "1999")]**
2. **//article//sec[about(//p, 'DEHOMAG')]**

Notice that the second fragment contains the full path including the "article" step.

Next each EvaluatablePathFragment is evaluated – the eval() method will return a set of nodes whose *contexts* match the full path for that fragment. For example fragment 2 above may return a node with the context:

**/article[1]/bdy[1]/sec[2]**

## 4.5 Merging Fragments

After each fragment is evaluated independently, we will have a list of node sets (one for each fragment) that must be merged. For example when merging the two sets from the above fragments, we will wish to include only those elements returned from the first fragment if they also have a descendent node contained in the set returned from the second fragment. In fact, what we need to return are elements with a context that matches the full path of the last fragment (in the case above they must have a context that matches //article//sec – the last named element in the context must be "sec"). What is meant by "including elements from the first fragment" is that the *SupportElements* for those *ReturnElements* in the first set will be added to a descendant *ReturnElement* (if it exists) in the 2nd set.

For example: let us say that the first set contains a ReturnElement with the context "/article[1]" and that ReturnElement has an attached SupportElement of "/article[1]/fm[1]/yr[1]" (for the purposes of this example assume that all contexts are in the same document.) Then let us say that the second set contains a ReturnElement of "/article[1]/bdy[1]/sec[2]". This element is supported by /article[1]/bdy[1]/sec[2]/p[3]. In this case the ReturnElement in the 2nd is a descendant of the ReturnElement in the 1st set – so we can merge the supports from the 1st ReturnElement into the supports of the 2nd and we will end up with a ReturnElement ("/article[1]/bdy[1]/sec[2]") that has 2 supports ("/article[1]/fm[1]/yr[1]" and "/article[1]/bdy[1]/sec[2]/p[3]").

When merging sets we must determine whether to do a strict merge or a union merge. For example if we need to merge the 2 fragments above, fragment 1 is "strict" – all elements that we merge from fragment 2 must also have an ancestor "article" element that contains a "yr" element for "1999".

The last fragment will always require a strict merge. This is because of the requirement stated above, that all elements returned by the query must have a context that satisfies the full path of the query.

However, a Union merge can be appropriate when we are merging two fragments where neither are the last fragment in the query, and both are non-strict (for example both only contain "about()" filters. In this case all ReturnElements will be retained, whether an element returned from the second fragment is a descendant of some element from the first fragment or not.

## 4.6 Support Elements

Support elements are elements that were found to contain at least one instance of a term that was specified in the filter. The element that contains this term must satisfy the full path for that filter including the context path.

In our example above the first filter (first fragment) looks for occurrences of the term "1999" in elements whose context matches the path "//article//yr". If we find that the term "1999" occurs in an element with the context "/article[1]/bdy[2]/sec[1]/p[1]" this is not a valid support for this filter. However, if we find a single occurrence of "1999" in the context "/article[1]/fm[1]/yr[1]" this would be a valid support.

Once we have removed all supports that do not represent valid supports (according to the filter), we then can create the return elements for this filter. In this case the return path is "//article" so the return element would have the context "/article[1]" with an attached support element with the context "/article[1]/fm[1]/yr[1]" and having one "hit" for the term "1999". It is possible that a return element contains more than one support element. For example, if within the same document we find another element with the context "/article[1]/fm[1]/yr[2]" that contains 2 hits on the term "1999" we would add another support element to the return element and record 2 hits on it. (This example is spurious as in the case of an equality constraint you actually only want to find one hit on the term. However it would make sense in the context of the "about()" filter).

## 4.7 Ranking

Previous works on document ranking in text retrieval are too numerous and diverse to mention in the INEX context. However, some relevant work has been done on ranking schemes of XML [7]. Many of them apply techniques used in classical Information Retrieval such the vector space model and apply them to structured documents, taking into account that relevance should be usually judged on a level smaller than that of a document

The approach we adopted in ranking was a multi-stage sorting process.

- First sort by f*ilter satisfaction*.
- For ReturnElements that satisfy the same number of filters - sort by number of distinct terms and phrases that were hit.
- For ReturnElements with the same number of filters satisfied and the same number of distinct terms - calculate a score based on total number of terms hit adjusted by a factor that penalises terms that are very common in the document collection.

### *5.7.1 Filter Satisfaction*

A ReturnElement is considered to have satisfied a filter where it is a valid ReturnElement for that filter, and it has a least one SupportElement that has recorded a hit for at least one term in the filter. A valid ReturnElement is one whose context matches the path expression of the filter.

In its simplest form, the filter satisfaction algorithm will rank higher a ReturnElement that has satisfied a greater number of filters. There are a number of refinements to this rule:

- Where two filters appear as *Predicates* to different *Steps* in the query expression (e.g. //article[//yr = “1999”] //sec[about(./, 'DEHOMAG')] ), each one of these filters that is satisfied will count towards the overall *filter satisfaction count.*
- Where two filters appear in the same Predicate and they are and-ed together (e.g. //article//sec[[//yr = “1999”] **AND** about(./, 'DEHOMAG')] ), each one of these filters that is satisfied will count towards the overall *filter satisfaction count*.
- Where two filters appear in the same Predicate and they are or-ed together (e.g. //article//sec[[//yr = “1999”] **OR** about(./, 'DEHOMAG')] ), if both filters are satisfied only one will be counted towards the overall *filter satisfaction count*.
- If any unwanted terms (prefixed by a minus) are hit in a SupportElement for the ReturnElement, then the *filter satisfaction count* will be reduced by a count of 2.

### *5.7.2 Distinct terms and phrases*

This algorithm is a second stage sort after the *filter satisfaction* sort. Where two ReturnElements have the same *filter satisfaction count,* the distinct terms algorithm is applied to determine their relative rank. Here we rank ReturnElements based on the number of distinct terms and phrases that they satisfy.

If a SupportElement has recorded hits for a particular term, its containing ReturnElement will have it’s *distinct terms and phrases count* incremented by one. Take for example the query:

**//article[about(.//st,'+comparison') and**

**about(.//bib,'"machine learning"')]**

Let us take the case where we have two ReturnElements that satisfy both filters. The first ReturnElement has supports that hit the terms “comparison” and “machine”. The second ReturnElement has supports that hit the terms “comparison”, “machine” and “learning”. In this case the second ReturnElement will be ranked higher. Note that it does not matter how many times each term is hit – it only matters if a term was hit at least once, or not at all.

The *distinct terms and phrases count* secondly takes into account the number of phrases that a ReturnElement has supports for. For example, take the query and the second ReturnElement we discussed above. If this ReturnElement also contained a support for the phrase “machine learning” - that is to say a context was found where the words “machine” and “learning” appear directly adjacent to each other – the *distinct terms and phrases count algorithm* will increment the count by one.

### *5.7.3 Scorer penalizes frequent terms*

The final stage algorithm of the 3 stage sort is only invoked where two ReturnElements have the same *filter satisfaction count* and *distinct terms and phrases count.* This algorithm calculates a score based on the total number of instances that terms were hit by SupportElements. The total number of hits for a term is normalized based on heuristic that takes into account how frequently that term occurs in the entire documents collection. This normalization factor is calculated as follows:

- *Hits*: Total number of instances that this term appears in the ReturnElements supports.
- *TF (TermFrequency)*: Count of number of times this term appears in total document collection

- *TFC (TermFrequencyConstant)*: A constant (determined using heuristics)
- *Score*: The ranking score for this ReturnElement
- *Terms*: The set of terms the score is based on
- i: Denotes the term
- *SM (ScarcityMultiplier)* = 1 + (*TF* / *TFC*)
- *Score* = ? $^{\text{i in } Terms}$ (*hits*$_i$ * (1/ *SM*$_i$))

## 4.8 Discussion on Ranking

Our overall ranking strategy was based on a series of heuristics.

### *5.8.1 Filter Satisfaction*

It is clear that our strategy places a high degree of importance to whether a particular collection of query terms are aggregated into one filter or if they are put in separate filters. For example, let us take the following two queries:

- //article[about(.,'clustering distributed') and about(., 'java')]
- //article[about(.,'clustering distributed java')]

Whilst these filters may appear logically equivalent, our filter satisfaction algorithm will mean that lists returned from each query formulation will vary significantly in how they are sorted. With the first query, the term "java" is raised to the same level of importance as that of both the other terms ("clustering" and "distributed"). By contrast, with the second query, a result that hits "clustering" and "distributed" (but not "java") will rank equal to a result that hits "distributed" and "java" (but not "clustering"). However, if the first query formulation is used the second result would be ranked higher as it satisfies two filters whereas the first result only satisfies one.

We believe this ranking strategy works well due to the psychology involved in creating these two filters. It can be inferred that when a query writer aggregates terms into one filter he/she considers all terms so aggregated of equal importance. In contrast, where a query writer puts terms in separate filters they are indicating that whilst each filter should be treated of equal importance, terms contained in separate filters are not necessarily of equal importance.

The second thing worth discussing about the filter satisfaction algorithm is the way it treats or-ed filters versus the treatment for and-ed filters. Let us take another two filters by way of example:

//article[about(.,'clustering) and

about(.,'distributed')]//sec[(about('java')]
//article[about(.,'clustering) or

about(.,'distributed')]//sec[(about('java')]

Further, let us assume we have *returnElement1* that hits the terms "clustering" and "distributed" and *returnElement2* that hits the terms "clustering" and "java".

In this case query 1 will rank *returnElement1*and *returnElement2* equal (both with a *filter satisfaction count* of 2). However query 2 will treat these quite differently. The *returnElement2* will still have a *filter satisfaction count* of 2 but the *returnElement1* will have a *filter satisfaction count* of only one.

Again we believe this makes intuitive sense. The second query construction implies that the user wants one of "clustering" or "distributed" to be hit – they don't care which – and if they are both hit then this is not as important as if "java" is also hit. It is interesting to note that the following query would be equivalent to the second query:

//article[about(.,'clustering distributed')]//
sec[(about('java')]

One final thing to note about this algorithm is how it treats unwanted terms (i.e. terms preceded by a minus sign). The algorithm is very harsh in how it treats the occurrence of such terms (by deducting 2 from the overall *filter satisfaction count*). However, We have found this works well in practice as the specification of such unwanted terms by the query writer appears to indicate a very strong aversion to that term.

### *5.8.2 Distinct Terms and Phrases*

The *distinct terms and phrases* algorithm is important in two respects:

- It places a greater importance on the number of distinct terms hit, than on the total number of instances that a term or terms are hit. (This can also be said about the *filter satisfaction* algorithm).
- Phrases are given prominence by the fact that they in effect count as an additional distinct term.

Let us consider the consequences of first point above. Take as an example the filter "//article[about(.,'clustering distributed java')]". Let us say that a ReturnElement records hits on the term "clustering" and "distributed"; both terms with 100 instances of this term occurring in the return's supports – a total of 200 recorded hits. Then let us take another ReturnElement that records just the one instance of a hit on each of "clustering", "distributed" and "java". It may surprise that this second ReturnElement will be ranked higher when it only recorded 3 separate hits versus the 200 of the first ReturnElement.

However, we believe this strategy has worked quite well in reality. What we have found that this in effect gives a greater prominence to those terms that do not occur frequently – that is it weights infrequent terms more heavily than frequent terms. This makes intuitive sense as an infrequent term that appears in a query is more likely to aid the precision of the recall than frequent terms. The more frequent a term is in the overall document collection the less value it has to determining the requirements of the user.

As regards the 2nd point above about giving phrases prominence, this should be self explanatory. Phrases occur much less frequently than individual terms, so it makes sense to treat them with a level of importance equivalent to the individual terms.

### *5.8.3 Scorer penalizes frequent terms*

Finally we discuss the algorithm that is invoked where the above two algorithms still cannot separate two equally ranked ReturnElements. It is only in this final stage algorithm that we take into account how “strong” the support is for a ReturnElement – that is how many instances of hits on terms have been recorded in a ReturnElement’s SupportElements.

As per our discussion for the *distinct terms and phrases* algorithm, here we also wish to penalize infrequent terms. The algorithm we developed to do this was refined by running a series of experiments and running our own assessment on the results to see if the modified algorithm improved the results. The *TermFrequencyConstant* gives us the ability to adjust the *normalization factor* for penalizing frequent terms.

## 4.9 Exceptions

Some INEX topics included conditions that could not be easily evaluated in the absence of external knowledge. For instance, a conditions such as **about[.//yr,2000].** Such a condition can be easily evaluated if a user, or an external schema can be consulted, in which the meaning of “about” in relation to <yr> can be determined. Furthermore, in practical terms, the implementation must take account of the type of the element (e.g, is it numeric or alphanumeric?).

The treatment of equality functions involving years (i.e. a "yr" tag) is straightforward: a string comparison is made between the value in the tag and the constant. However, the treatment of inequality functions (i.e. those involving inequality operators "<", ">", "<=", ">=") is more complex. The greater than operator is undecideable as the upper range of year values to search the index for is unbounded. The less than operator may be decideable if we take the year 1 as the lower bound - but in this case the practical consequence of having to search the index for upwards of 1,990 terms is that we need to define a more reasonable lower bound. As such, we allowed for the lower and upper bound of year terms to be configured via our configuration file. This results in a managable range of year terms for which we have to search the index for any reasonable year based inequality predicate. The “about” was also defined in a configuration file (3 years either side of specified “about” year).

## 5. Experimental Results

The system was only designed for Content and Structure queries (CAS). Only the <Title> element was used in topic evaluation. The system was not designed to take advantage of information contained in the <Description> and <Keywords> elements of a Topic.

## 5.1 Strict Content and Structure

The best results were obtained with the SCAS query and strict quantization metric. The average precision was 0.26 (the submission was ranked **3rd**.) With the Generalized quantization metric the system was ranked 8th. These results are somewhat surprising given that we only used the <Title> element of a topic. One would have expected the use of additional keywords from the <Description> and <Keywords> elements to assist retrieval and ranking.

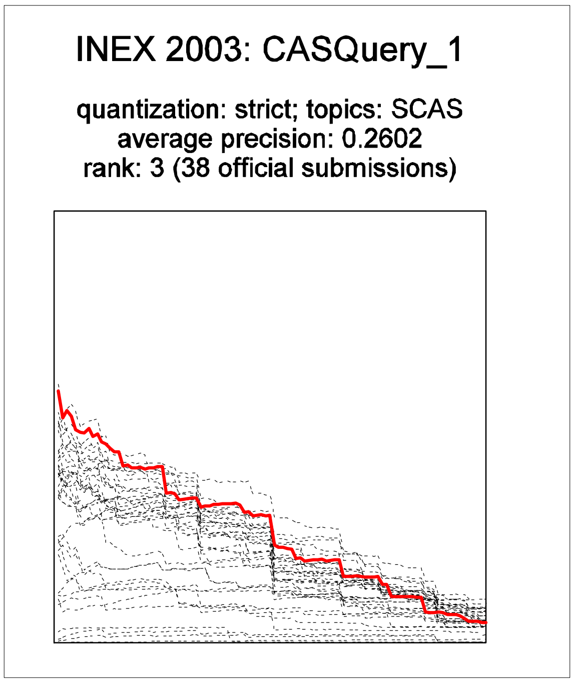

Figure 2: Retrieval results for Strict Content and Structure (SCAS) topics, quantization Strict

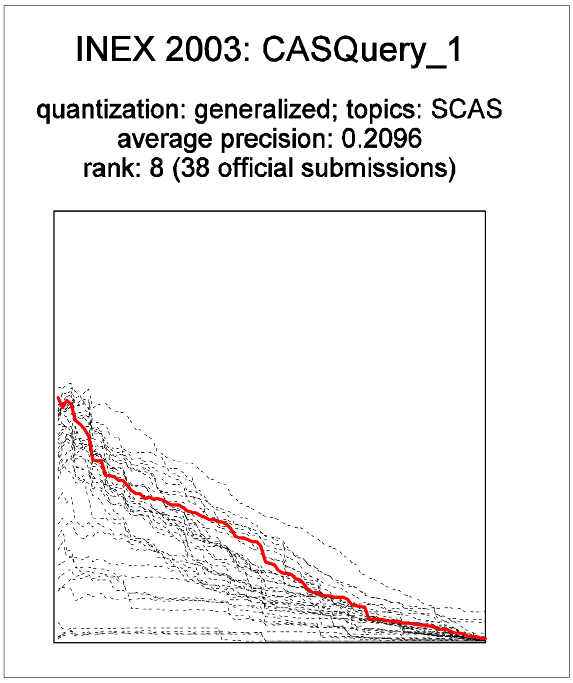

Figure 3: Retrieval results for Strict Content and Structure (SCAS) topics, quantization Generalized.

### 5.2 Vague Content and Structure

Results for VCAS are not available at the time of writing this paper.

## 6 Discussion

There is no question that the formulation of the <Title> element of an XML topic at INEX 2003 is not end user oriented. However, it does allow for exact specification of structure and content constraints. We were able to implement a search engine that evaluates CAS <title> expressions with good accuracy and reasonable response time. Furthermore, we were able to construct the search engine on top of a generic XML inverted file system. This allows the application of the system to XML collections without explicit reference to the underlying XML Schema (or DTD). It seems however that in the definition of INEX CAS Topics the authors did not always specify the intent of the topic (as evident in the topic's narrative) in an accurate manner. This ultimately must have lead to low precision (across all submissions from all participants).

We were not able to solve the problem in a completely generic fashion because some topics' structural constraints could not be easily interpreted in a generic manner (e.g treatment of about conditions over <year>). This problem can be overcome to some extent with the use of an XML Schema in future evaluations at INEX.

## REFERENCES

[1] Proceedings of the First Workshop of the Initiative for the Evaluation of XML Retrieval (INEX), DELOS Network of Excellence on Digital Libraries, ERCIM-03-W03

[2] Programmar™ Developers Toolkit, NorKen Technologies, Inc., http://www.programmar.com/

[3] "XML Path Language (XPath) Version 1.0" http://www.w3.org/TR/xpath

[4] Govert, N. (2002), Content-oriented XML Retrieval with HyRex, Proceedings of the First Workshop of the Imitative for the Evaluation of XML Retrieval, Schloss Dagstuhl, Germany, December 9-11, pp 26-32a

[5] Ceponkus, A. (1999), *Applied XML: A Toolkit for Programmers*, John Wiley & Sons, New York

[6] Zobel, J. Moffet, A., and Ramamohanarao, K (1995). Inverted files versus signature files for text indexing. Tech Rep. CITRI/TR-95-5, Collaborative Information Technology Research Institute, Department of Computer Science, Royal Melbourne Institute of Technology, Australia, July

[7] Proceedings of the First Woorkshop of the Initiative for the Evaluation of XML Retrieval (INEX), Ed: Fuhr, Goevert, Kazai, Lalmas, 2002.

[8] Schlider T. & Meuss, H. (2002), Querying and Ranking XML Documents, JASIST, 53(6), pp. 487 - 503

[9] Fullerr, M. Mackie, E., Sacks-Davis, R. and Wilkinson, R. (1993), Structured Answers for a large Structured Document Collection", in Proceedings of ACM SIGIR '93, Pittsburg, PA, 204-213, June 1993

[10] Lamas, M. (1997), Dempster-Shaferâ€™s Theory of Evidence Applied to Structured Documents: Modelling Uncertainty. in Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 110 - 118, July 1997

[11] Crestani, F. Lamas, M., Van Rijsbergen, C. J., Campbell, I., Is This Document Relevant? Probably. A Survey of probabilistic Modells in Information Retrieval. ACM Computing Surveys, 30(4), 2001 Strzalkowski, T., Prezez-Carballo & Marinescu M. (1996)