# Distributed XML Information Retrieval

Wayne Kelly
Centre for Information Technology Innovation
Faculty of Information Technology
Queensland University of Technology
GPO Box 2434 Brisbane Q 4001 Australia
w.kelly@qut.edu.au

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

Tony Sahama
Centre for Information Technology Innovation
Faculty of Information Technology
Queensland University of Technology
GPO Box 2434 Brisbane Q 4001 Australia
t.sahama@qut.edu.au

Wengkai Loke
Centre for Information Technology Innovation
Faculty of Information Technology
Queensland University of Technology
GPO Box 2434 Brisbane Q 4001 Australia
w.loke@qut.edu.au

## ABSTRACT

In this paper we describe the implementation of a distributed search engine for XML document collections. The system is based on a generic P2P collaborative computing framework. A central server coordinates query and search results distribution. The server holds no documents nor does it hold any indexes. The document collection is distributed amongst multiple PC based workstations, where it is also indexed and searched. The system is scalable to databases several orders of magnitude larger than the INEX collection, by using a system of standard networked PCs.

**Keywords:** P2P, INEX, XML, Distributed Database, Information, Retrieval, Inverted File, XPath, Assessment, Evaluation, Search Engine

## 1. Introduction

Web search engines such as Google are enormously valuable in allowing ordinary users to access information on a vast array of topics. The enormity of the information being searched and the massive number of clients wishing to make use of such search facilities means, however, that the search mechanisms are inherently constrained. The data being searched needs to be a priori indexed. Searching is limited to finding documents that contain at least one occurrence of a word from a list of words somewhere within its body. The exact relationship of these words to one another cannot be specified. These limitations mean that it is often difficult to specify exactly what you want, consequently clients are overwhelmed by an avalanche of query results – if users don't find what they are looking for in the first couple of pages of results they are likely to give up.

XML documents contain rich structural information that can be used by information retrieval system to locate documents, and part thereof, with much greater precision than text retrieval systems can. However, systems capable of searching XML collections by content are typically resource hungry and are unlikely to be supported extensively on central public servers for some time to come, if at all.

Peer to Peer (P2P) file sharing systems such as KaZaA, Gnutella and Napster enable documents to be searched and accessed directly from end user's PCs, i.e., without needing to *publish* them on a web server, but again the indexing for retrieval is a priori. This is fine if you are searching based on well defined metadata keys such as song title or performer, but not if you are trying to search based on the content of the data.

The greatest degree of search specificity is achieved if the search engine can potentially access the content of the entire collection for each and every query. Obviously this is infeasible for huge document collections such as the entire WWW. If, however, we limit ourselves to smaller collections such as documents archived by a "community" of individuals that are collaborating on some project or share some common interest, then such a precise Information Retrieval paradigm is feasible and highly desirable.

The P2P framework that we propose is based on *search agents* that visit the workstations of participating individuals to perform custom searches. Individuals wishing to perform a search can choose from a library of "standard" search agents, or they can implement their own

agent that implements an arbitrarily sophisticated search algorithm. The agents execute on the individual workstations within our P2P host environment that "sand-boxes" them, preventing them from doing "harm" to the workstations and allowing the workstation owners to control exactly which "resources" can be accessed. Resources potentially accessed include files, directories and databases. The key advantages of our system compared to web search engines such as Google are:

- Arbitrarily sophisticated algorithms can be used to perform highly selective searches, since the query is known before the actual document collection is scanned.
- The documents don't have to be explicitly published to a central server – they are accessed in place. This saves time and effort and means that working documents can be made immediately available from the time they are created, and work can continue on those documents locally while still being externally accessible.
- Volunteers have the option to only partially publish documents. This means they allow a client's search agent to examine their documents, but they limit the response that such search agents can return to the client. The response could be as limited as saying "Yes - I have a document that matches your query". In most cases, the agent will return some form of URL which uniquely identifies the matching document, but our framework doesn't in itself provide a mechanism for the client to retrieve that document from the volunteer. The exact mechanism by which such documents are retrieved is beyond the scope of this paper, but it could for example be a manual process, whereby the owner of the volunteer workstation will access each such client request based on the identity of the client and the document being retrieved. This might happen, for example, in a medical setting with doctors requesting patient records from other doctors, or in a law enforcement setting with police agencies requesting criminal histories from other jurisdictions.

The remainder of this paper is organized as follows. In section 2 we describe the system underlying the distributed search engine. In section 3 we describe the XML search engine that is distributed and executed by search agents on the distributed database. In section 4 we discuss the results of testing the systems against the INEX collection. In section 5 we discuss and summarize the lessons learnt from the INEX exercise.

## 2. System Architecture

Our system is termed P2P in that the actual searching is performed on peer nodes. The internal architecture of our system is, however, client/server based - for a number of reasons. The underlying architecture of our system is illustrated in Figure 1. The client PCs that make up the "leaves" of system belong to the individuals in the community and can play two distinct roles; they can be a *searcher* or they can *volunteer* to be searched. A *searcher* is a PC that submits queries to the system. The *volunteers* are the PCs on which the documents reside and on which the queries are processed. Individual PCs can play either or both of these roles at various points in time. PCs volunteer themselves to be searched typically only when they are otherwise idle. This is a form of cycle stealing, as the execution of the search agents may consume considerable CPU time and memory bandwidth of the machine while it is running.

The clients of the system - the searchers and the volunteers come and go over time; the *search server* is the only part of the system that remains constant. It acts as a central point of contact for searchers wishing to submit queries and for volunteers willing to be searched. It also acts as a repository for queries waiting to be processed and query results waiting to be retrieved. At the point in time when a searcher submits a query, there may be some volunteers "currently connected" to the server that would be willing to process that query immediately. In such a case some results may be able to be returned to the searcher almost immediately (allowing of course, for the time to perform the search on the volunteer machines - which can be arbitrarily long depending on the complexity of the search algorithm and the size of the document collection being searched on each PC).

Often, however, the relatively small set of set of volunteers that are currently connected, will either produce no results for the query, or at least will produce no results that are satisfactory to the searcher (note that this is made more probable by the high degree of query specificity that is possible with an agent based search framework). In such a case, we assume the searcher will often be willing to wait (minutes, hours, days or perhaps even weeks) for other volunteers to connect to the system and hopefully contribute interesting new results. This is the key difference between our distributed search engine and traditional cycle stealing systems. In a traditional cycle stealing system, all volunteers are considered equal – once a computational task has been carried out by one machine there is no point if her volunteer machine repeat that

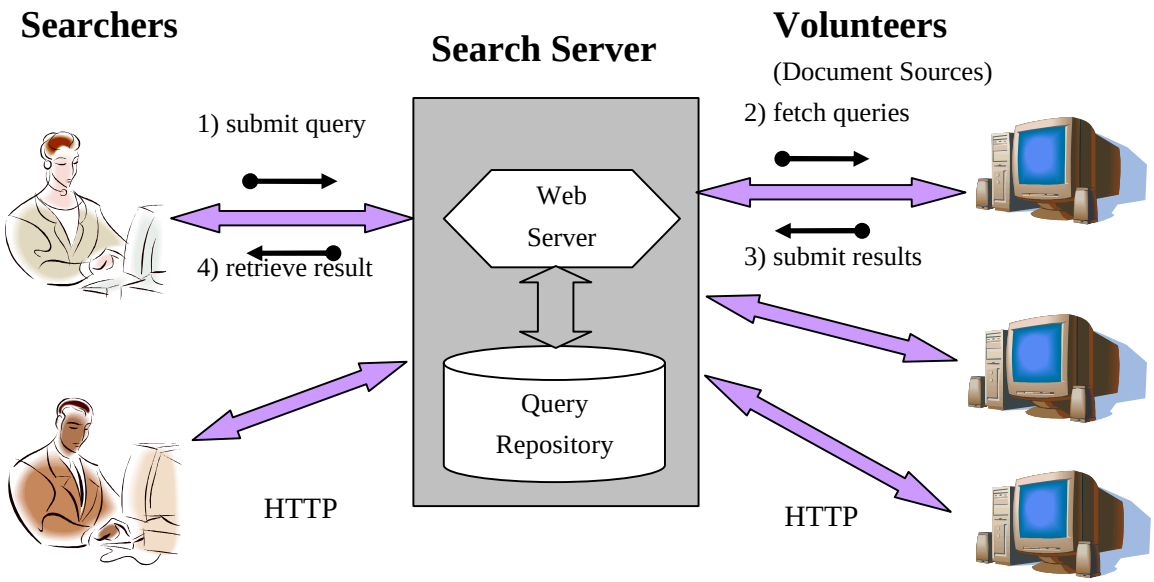


**Figure 1: System Architecture**

same computation. In our distributed search system, however, each PC is assumed to archive a different set of documents – so even if a query has been processed on one volunteer, it still makes sense to keep that query around for other volunteers to process when they connect later.

Having a query and results *repository* allows the submission of queries and results to be separated in time from the fetching and processing of those queries and results. Having a central server means that once a client has submitted a query, it can disconnect from the system, and only reconnect much later when it expects to find a significant collection of results. More importantly, the wide spread use of corporate firewalls will often mean that PC's performing searchers cannot directly communicate with many potential volunteers and vice versa. Having a central server that is able to receive HTTP requests from anywhere on the Internet has the effect of providing a gateway for searchers and volunteers to work together who would otherwise be unable to communicate. Note, installing a web server on all searcher and volunteer PC's would *not* achieve the same effect – a HTTP request message generally can not be sent to a machine behind a firewall, even if that machine hosts a web server.

The search server exposes interfaces to searchers and volunteers as SOAP web services transported using HTTP. Searchers can submit queries and fetch results and volunteers can fetch queries and submit results. All communication is initiated by either the searchers or the volunteers, and connections are not left open; i.e, the server can't push either queries or results to the searchers or the volunteers - they must request them. From the volunteer's perspective, the server is stateless. The server maintains *neither* a list of currently "connected" volunteers, nor a list of all potential volunteers. Anyone can volunteer at any time (subject to any authentication that the server may implement to ensure that the volunteer is a member of "the community"). When a volunteer connects to the server (after having been "disconnected" for a period of time) it receives a list of all queries that have been submitted to the server since that volunteer last connected. Each volunteer is responsible for keeping a "time-stamp" (in reality a sequence number allocated by the server) that represents the point in time at which that volunteer last requested queries from the server. In this way, the server is spared from maintaining information specific to each volunteer yet is able to respond to requests from individual volunteers in a personalized manner.

The time period that a query remains on the server is determined by a number of factors. Firstly, the searcher can specify a "time-to-live" when they submit the query. This may be overridden by the server which may dictate a system wide maximum "time-to-live" for all queries. Individual volunteers may also implement their own policies, such as refusing to process queries that are older than a certain date. Finally, the searcher can manually retract a query from the server as soon as they have received satisfactory result(s) to their query or if they realize that the query was incorrect or too inexact.

## 3. The XML Search Engine

The search engine is based on an XML inverted file system, and a heuristic approach to retrieval and ranking. These are discussed in the following sections.

## 3.1. The XML Inverted File

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] |
| 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] |

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.

## 3.2 Normalized Database Structure

The structure of the database used to store the inverted lists is depicted in Figure 2. 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 inverted list thus obtained is *Joined* (SQL) 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.

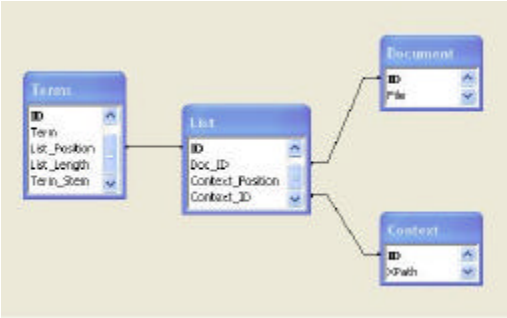


Figure 2: Database Schema for the XPath based Inverted XML File.

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. Phrases and other proximity constraints can be easily evaluated by using the ***Context_Position*** of individual terms in the ***List*** table.

With this normalization the database size was reduced to 1.6GByte and within the Microsoft Access limits. This is of course a trade-off in performance since costly join operations may be necessary for the more frequent terms.

## 3.3 Searching the Database

The database structure enables the identification of inverted lists corresponding to individual terms. Each term that appears in a filter of an INEX <Title> element has an associated Xpath context. Terms that appear in a <keywords> element of a topic have the default context of /article. With simple SQL statements it is easy enough to retrieve inverted lists for terms that satisfy a filter.

### *3.3.1 SCAS topics*

Our search strategy for SCAS topics consists of several steps, as follows.

We start by fragmenting the INEX <Title> element into several sub-queries, each corresponding to a filter on the path. So, for instance:

**<title>//article[about(.//st,'+comparison')/**
**bm[about(.//bib,'machine learning')]</title>**

is transformed to a set of 2 individual queries:

**S|//article|//article//st|+comparison**
**R|//article/bm|//article//bm//bib|machine learning**

This formulation identifies two sub-queries, each with 4 parts delimited by a '|'. The **S** denotes a support element and the **R** denotes a Returned Element. The support element has the Xpath signature **/article**. The return element has the XPath signature **/aticle/bm**. The support element filter looks for elements with the Xpath signature **//article//st,** containing the term "**comparison"**. The returned element filter looks for elements with the Xpath signature **//article/bm//bib,** containing the phrase "**machine learning**".
Strict compliance to the XPath signature of the various elements is enforced. However, this is moderated by the use of equivalent 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**

Each of the elements is scored in the following way – we count the number of times that each term in the filter is found in the element. If more than one term is found then the term counts are multiplied together. This has the desired heuristic that elements containing many search terms are scored higher than elements having fewer search terms.
The score of a returned element is the sum of the scores of all its support elements. So in the example above, the score of a **//article/bm** element is the sum of all the corresponding **//article//st** elements (within the same <article>) and all **//article/bm//bib** elements (within the same <article> and same <bm>). At one extreme a returned element may be supported by numerous elements from all filters. At the other extreme it may only have support in one term of the returned element filter. We accept all such return elements as candidates for results. However, the returned elements are sorted first by the number of support filters that they satisfy and then by their score.
Topics that make use of AND clauses and OR clauses in the <Title> are handled by generating separate query for each clause. We do not distinguish between AND and OR and effectively allow ranking to take care of it. The heuristic justification is that if all terms appear then the score should be higher regardless of whether AND or OR were used. Also, if AND was specified, but only satisfied by some of the terms, we still want the partially matching elements as potentially valid results – after all, this may be the best that we can find.
The <Keywords> element of topic is also used – it defaults to a query on the entire <article> and considered a support to all returned elements within the same article.

### *3.3.2 VCAS topics*

The VCAS queries were treated in exactly the same manner as SCAS queries, except that we expanded the equivalence tag interpretation.

 ılent 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**

### *3.3.2 CO Topics*

The CO topics were handled in the same manner as CAS topics. However, all terms from both the <Title> and <Keywords> elements of the CO topic were combined to form a single query – after removing duplicate terms. The return element was assigned the default XPath signature //* which means that any element in the article was returnable (subject to support). For instance, topic 91 –

**<title>Internet traffic</title>**
**<keywords>internet, web, traffic, measurement, congestion </keywords>**

is transformed to the following query:

**R|//*|//article|Internet,traffic,web,measurement, congestion**

Every element with the context of //article (this includes descendents) and which contains at least one of the terms in the query is suitable for return. However, since only leaf nodes in the XML tree contain terms (with very few exceptions) there is a need to associate a score with other non-leaf elements in the tree in order to qualify them for selection. The search engine propagates the score of matching elements upwards, recursively, to ancestor nodes, in the following manner. If an ancestor has a single child it receives half the child's score. If it has multiple children it receives the sum of their scores. In this manner, for instance, a section with multiple scoring paragraphs receives a score higher than any of its paragraphs and will be ranked higher. A section having only one scoring paragraph will be ranked lower.

### *3.3.4 Selection by Year*

Selection by year was treated as an exception. The search engine expands conditions with respect to years to allow for a range of years. It allows up to 5 years below for a Less Than condition, up to 5 years above for a Greater Than condition, and 2 years either side for an about condition. Equality is treated strictly. This is necessary for two reasons. The inverted list structure does not support range queries so it is necessary to translate such conditions to explicit values that can be searched. It is also not possible to interpret the *about* condition over <year> without some pre-conceived idea of what might be a reasonable year range.

### *3.3.4 Term expansion*

The search engine can optionally expand keywords in one of two ways. It can perform plural and singular expansion, or it can use the full porter stem (pre-stored in the database). In the case of phrases, the program also attempts to construct an acronym. So for instance, the phrase "Information Retrieval" generates the additional term "IR". A common writing technique is to introduce an acronym for a phrase and thereafter use the acronym for brevity. For instance, at INEX, we defined "Strict Content and Structure" as "VCAS". Subsequent references are to VCAS only. So the idea here is to try and guess acronyms. We use several simple rules that attempt to manipulate the phrase initials to construct a few acronyms. If an acronym thus generated is found in the inverted list it is used as an additional term.

## 4. Results

Two aspects of the system were tested. The precision/recall values were measured through the standard INEX evaluation process. The performance of the distributed search engine was also tested on a distributed database.

## 4.1 Performance

The system was tested as a stand alone search engine in a single PC and on a distributed configuration. On a single PC (Pentium 4, 1.6GHz, 500MB RAM) the search times for topics varied between 5 seconds and one minute, depending on the number of terms and their frequency in the database.

The database can be distributed in a logical manner by placing each of the 18 journals on a different PC. Each search engine was set to return the N best results. We used a threshold N=100, but this is a run-time argument. The communications overhead of the system is about 5 seconds (pretty much fixed, given a reasonably fast connection.) The search over a single journal is very quick and takes less than 3 seconds. The INEX collection can thus be searched in less than 10 seconds even for the most elaborate topics. The total search time is pretty much upper limited by the longest search time on any of the distributed components. Nevertheless, results arrive asynchronously, so the user can view early results before the entire distributed search is complete.

The system scales up well. If the full database is duplicated on several PCs the search time is virtually constant – as long as the number of results returned is reasonably capped.

Results are ranked independently by each distributed search component. Consequently, the results can be displayed in order, either globally, or within each Journal. A difference between the single complete database and the distributed database results can arise if there are useful results in one journal that are ranked below the allowed threshold N. However, this difference will only affect the lower end of the ranked list and in any case this problem can be easily circumvented. An obvious variation is to determine the return threshold by rank rather than by count. In this manner poor results can be avoided while better results are allowed to arrive in larger numbers from fruitful searches of distributed database compartments.

## 5.2 Precision/Recall

The better results were obtained in the SCAS track with plural/singular term expansion. It scored an average precision (generalized) of 0.195 (rank 12/38). The Porter stemming expansion of terms produced somewhat lesser results with an average precision of 0.186. Without term expansion the results had an even lower score with an average precision of 0.174.

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

In the CO track results were similar. The better results were obtained with full Porter stemming, with an average precision (generalized) of 0.0525 (rank 14/56). Somewhat lesser, but essentially similar results were obtained with plural/singular expansion with an average precision of 0.0519. Without term expansion the average precision was 0.0505.

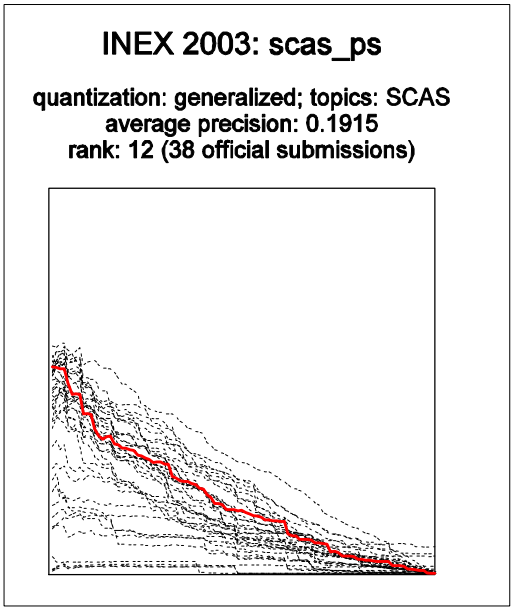


Figure 3: Plural/Singular expansion

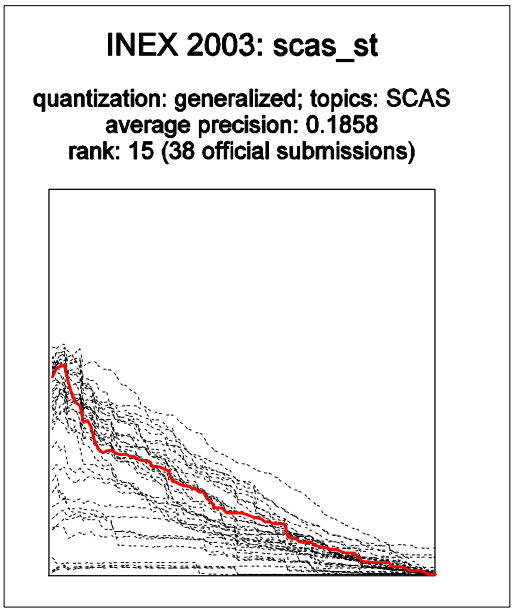


Figure 4: Full Porter stemming

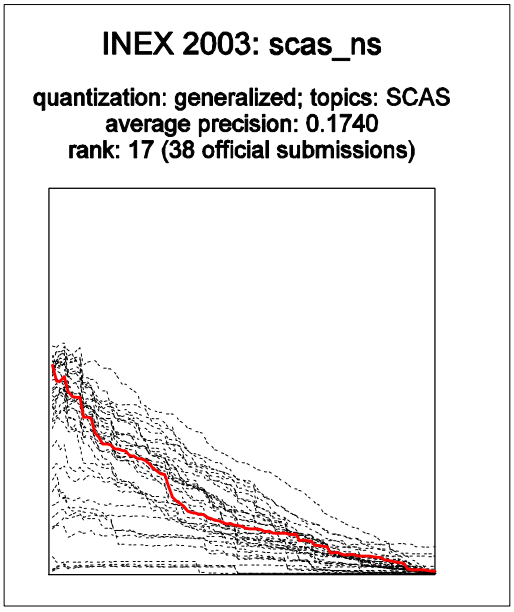


Figure 5: Without Term expansion

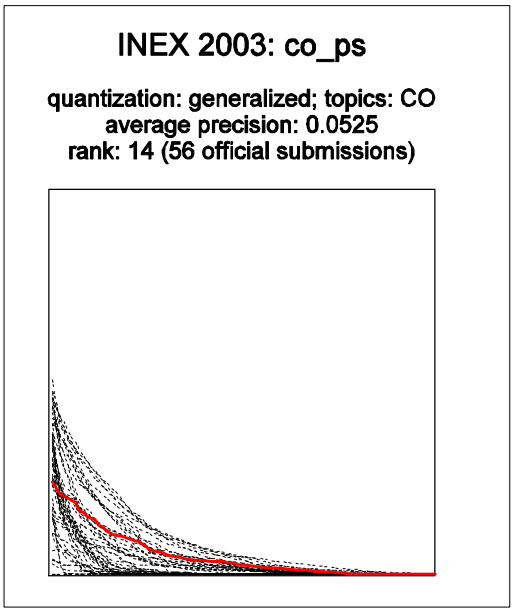


Figure 6: Full Porter stemming

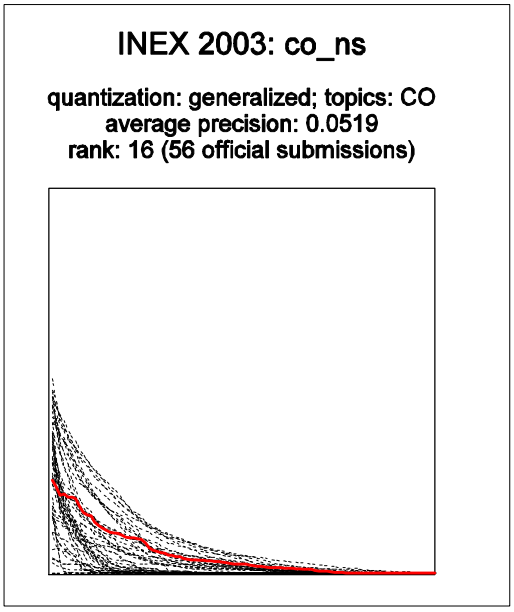


Figure 7: Plural/Singular expansion

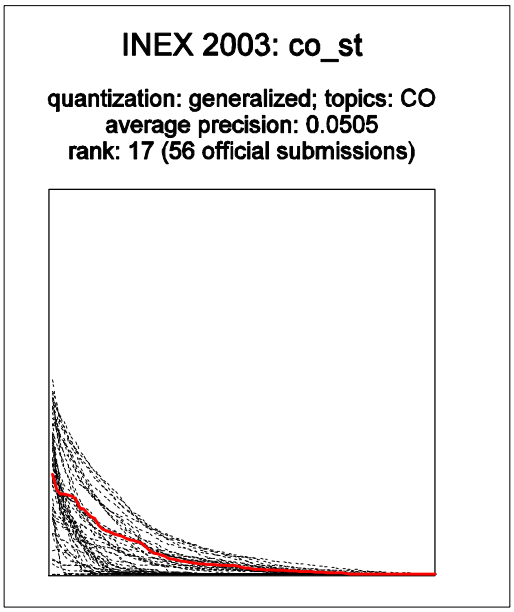


Figure 8: Without Term expansion

## 5. Discussion

The search engine that was developed and tested performs reasonably well in terms of precision/recall. It performs very well in terms of speed, and scales almost linearly.

Inspection of our results suggests that while the system was able to retrieve the most significant <article> elements, it fell short in terms of ranking the various descendents. With CAS queries the loose interpretation of AND, OR, and equality constraint might have contributed to violations of topic <title> XPath constraints leading to selection of undesirable elements. With CO queries the ranking heuristics that we used were generic. We only took account of abstract tree structure considerations. It might have been advantageous to also apply heuristics that are specific to the INEX collection and perceived intent of topic authors (in general, not specifically). For instance, paragraphs might be better units of retrieval than sections. More analysis and experimentation with ranking is required.

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