# Searching Structured Documents

Andrew Trotman

*Department of Computer Science, University of Otago, Dunedin, New Zealand. andrew@cs.otago.ac.nz*
*Research conducted at: National Center for Biotechnology Information, Bethesda, Maryland, USA.*

**Abstract**
Structured document interchange formats such as XML and SGML are ubiquitous, however information retrieval systems supporting structured searching are not. Structured searching can result in increased precision. A search for the author "Smith" in an unstructured corpus of documents specializing in iron-working could have a lower precision than a structured search for "Smith as author" in the same corpus.

Analysis of XML retrieval languages identifies additional functionality that must be supported including searching at, and broken across multiple nodes in the document tree. A data structure is developed to support structured document searching. Application of this structure to information retrieval is then demonstrated. Document ranking is examined and adapted specifically for structured searching.

*Keywords:* Structured Information Retrieval; Indexing and Searching; Vector Space; Boolean Searching; SGML and XML

## 1. Introduction
When documents were simple text files, queries could only be asked of the entire document. Today XML (Bray, Paoli, & Sperberg-McQueen, 1988) and SGML (ISO8879:1986) have become popular. Both XML and SGML encourage the markup of semantic documents elements. The title should be tagged as such. The authors should be tagged as such and so on. Taking advantage of this structure has two gains: increased functionality and increased precision (Schlieder & Meuss, 2002).

Work on increasing functionality is evidenced by the proliferation of XML query languages such as XIRQL (Fuhr & Großjohann, 2001), XQL (Robie, Lapp, & Schach, 1998), XML-QL (Deutsch, Fernandez, Florescu, Levy, & Suciu, 1998), and ELIXIR (Chinenyanga & Kushmerick, 2001). These languages identify two kinds of search, structure searching and content searching.

Structure searching finds document that have a given structure or substructure, for example "find all documents that have an address tag within an author tag". Searching for the existence of a tag structure is analogous to searching for the existence of search terms (Schlieder & Meuss, 2000).

Content searching is the IR search. Find all documents that contain the word "Seuss". When structure is available it becomes possible to build content-based searches using structure. "find all documents containing Seuss as the author".

XML query languages have been criticized because formulating a query requires intimate knowledge of the document structure (Schlieder & Meuss, 2002). This may well be true, but they do identify the ways structural paths can be specified.

Using XIRQL (Fuhr & Großjohann, 2001) notation, a query for "heading" searches for the existence of heading tags. A query for "body//heading" searches for a heading within a body. A search for "body/heading" searches for a heading as an immediate child of body. If a specification starts with a "/" it is relative to the root of the document.

The path specification constructs apply not only to searching for structure, but also to searching for content. There are four possible ways to search:

Search for a term
Search for a term in a tag
Search for a term in a partially specified branch of the document tree
Search for a term in a fully specified branch of the document tree

Although such queries can be specified, resolving the query against a database can be complex. It is conventional to print species names in italics, it is therefore reasonable to expect all instances of *E.coli* to appear marked up. A title might be "*E.coli* inquiry calls for stricter laws on selling meat**"**. The markup might be:

<title><organism><genus>E.</genus><species>coli</species></organism>inquiry calls for stricter laws on selling meat</title>

The query "coli within organism" should match this title even though the tag specified is not a leaf of the document tree. The query "phrase 'coli inquiry' within title" should also match this document even though the query lies across tag boundaries.

Research into query languages has progressed more quickly than research into implementing such languages. Implementations of structured search systems exist using relational databases (Beitzel, Jensen, Grossman, Ingham, & Lewis, 2001; Chinenyanga & Kushmerick, 2001), object databases (Shimura, Yoshikawa, & Uemura, 1999) and information retrieval systems (Burkowski, 1992; Lee, Yoo, Yoon, & Berra, 1996; Meuss & Strohmaier, 1999; Schlieder & Meuss, 2002; Thom, Zobel, & Grima, 1995), but each has limitations. Only IR systems support relevance ranking, however most do not fully support the query language constructs.

## 2. Related work

Lee et al. (1996) discuss a number of representation schemes for structured data and suggest that Inverted Index for All Nodes without Replication (ANOR) performs best. A corpus of structured documents is parsed into a single document tree. This document tree is then interpreted as a k-way virtual tree (some nodes may not exist) and each node given a unique identifier. Each term is then stored at the node which is the lowest parent of all occurrences of that term. This way each term is stored against only one node in the tree. The postings for a given term in a given record thus form the ordered pair (document-id, node-id).

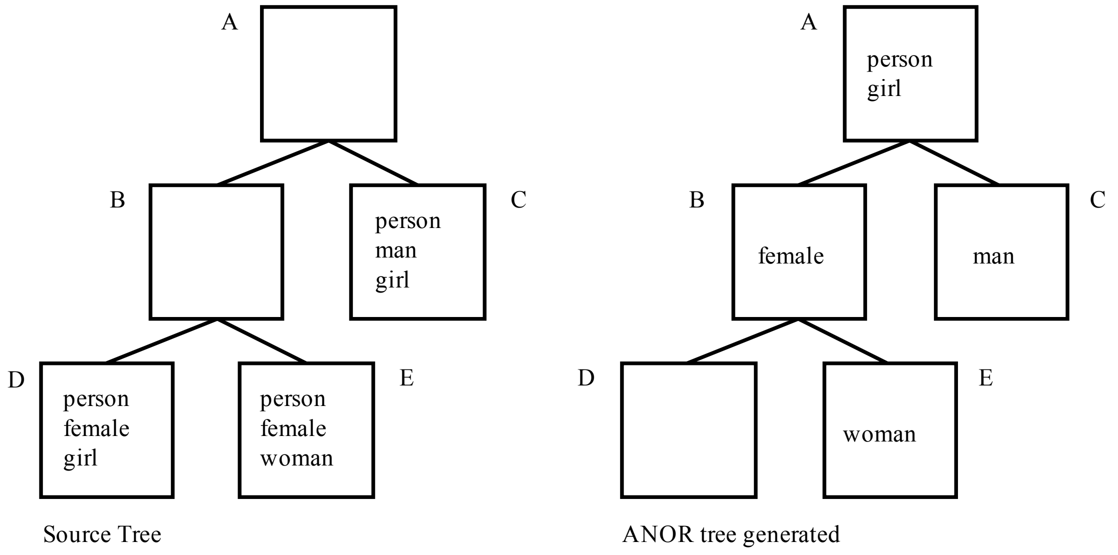

Figure 1: The ANOR tree incorrectly promotes girl to lie at node A implying existance at node E

The ANOR encoding is lossy. Term locations in the indexes are not the location in the original document; they are the lowest parent of all occurrences. Consequently, terms can lie in the encoded tree in locations in which they do not lie in the data. Such an example is given in figure 1. When a user issues the query "girl in E" the result is true, however it should be false. Girl has incorrectly been promoted to node A because it is the lowest parent of all occurrences of the term girl, even though girl does not appear at node E.

Shin et al. (1998) suggest a system called the Bottom Up Scheme. BUS stores terms only at the leaves of a corpus wide document tree (for example the source-tree in Figure 1). Nodes in the document tree are encoded using a similar technique to that of Lee et al. (1996). Each instance of each term is stored in the inverted files along with the node-id for the leaf node. Searching at any node in the document tree is supported through a bottom-up tree walk. Term frequencies are accumulated during walk. Once the walk is complete relevance calculations can be accurately determined from the accumulated occurrence counts. Searches at the root of the document tree are supported by a bottom up search of every tree node. To execute such a search requires loading and parsing many inverted lists.

Thom et al.(1995) suggest storing document tree paths directly in the postings. If each unique tag is given an ordinal identifier, each path through the document-tree can be represented as a string of integers. Any term lying in M places in the document tree can be represented by the tuple <docid, path length, path, count=M, offset 1, offset 2, …, offset M> where offset is the offset in indexable terms from the beginning of the specified path. Thom et al. report a major limitation to their encoding – it is unable to resolve proximity queries across tags. It cannot resolve the query "phrase 'coli inquiry' in title" in the earlier example because stored offsets are from the beginning of the path in which the term lies.

Meuss and Strohmaier (1999) also suggest storing the path directly in the indexes. A single document tree is constructed for the corpus, and each node given an ordinal number (from 0 through M). For any term in any location in the tree, the path to that term is represented as a bitstring of length M. If the term lies under a given node, a 1 bit is stored in the bitstring at the ordinal node location. Elsewhere a 0 is stored. A compressed form of the bitstring is stored with each posting. To determine if a given term lies at a given location in the document tree, the posting is read and the bitstring checked. These bitstrings can become very long so Meuss and Strohmaier suggest forfeiting recall for compression

Schlieder and Meuss (2000, 2002) examine indexing and searching in XML documents. Each tag in the DTD is given a unique identifier. Each occurrence of each term is stored in multiple inverted lists, one for each tag in which the term occurs. They demonstrate how document structure can be deduced from the indexes. If a given occurrence lies in two inverted lists, one must be the parent of the other. In order to determine which is the parent requires index augmentation. They augment with three identifiers, one of which is the preorder number of the node in the cross-corpus document tree. This requires complete knowledge of the structure of all indexable documents in advance.

Kotsakis (2002) represents the structure of XML documents as a tree whose nodes are given ids on an "as encountered during indexing" basis. Each posting is a pairing of record id and node id. This technique does not require the structure in advance, so database addition and deletion are supported. The Kotsakis method generates a separate inverted file for each leaf in the tree. Lee et al. (1996) suggest multiple index encodings are inefficient in storage as duplication could be vast.

Fuhr and Gövert (2002) suggest building a separate document tree for each document. Each node in each tree is given a unique path handle – a concatenation of document number and an ordinal encoding of tree location. Strings of these handles form the postings in the inverted lists. As the document tree is different for each document, the same path in multiple documents is likely to be represented by multiple identifiers. It is therefore necessary to store all trees in memory while searching. This is achieved through compression. Although these indexes are shown to compress very well, they are also shown to decompress inefficiently.

Burkowski (1992) and Dao (1998) suggest modelling the corpus as a series of contiguous extents. As a document is parsed each occurrence of each term is given a unique ordinal number. Each tag is represented as an extent. The start of the extent is the ordinal number of the first term inside the tag. The end of the extent is the ordinal number of the last term inside the tag. A structured query is represented as a series of successive filters. First find the postings for the term, filter it to lie within the given tag, and then filter to lie within given documents. Loading these filters from disk can take considerable time, as they can be larger than the postings for the searched terms. Thom et al. (1995) suggest the postings for a frequently occurring tag such as "paragraph" may take up 2-5% of the indexes.

The techniques presented thus far fall into two classes: inextensible solutions and extensible solutions. Inextensible solutions require advance knowledge of the document tree of all documents before indexing (documents with a different structure cannot be added). Such knowledge may not be available. Extensible solutions forfeit recall (Meuss & Strohmaier, 1999), forfeit functionality (Thom et al., 1995) or are required to perform excessive disk I/O (Burkowski, 1992).

## 3. Proposed Method

### *3.1. The Field Stream*

A document is a linear stream of characters. This character stream is converted into a tokenized term stream by breaking at separators such as spaces and commas.

If multiple documents are in the same input stream a reserved token is inserted into the tokenized term stream to signal the start of a document, and another to signal the end of a document. For streams containing structured data, field start and field end tokens are also inserted.

An example field stream might look like this:

<record><f1>term1</f1><f2>term2<f3>term3</f3></f2></record><record><f1>term4</f1></record>

For hierarchical data, sequences of field start tokens unbroken by field close tokens are generated. In the example above, f3 is inside f2, which is parallel to f1. Both records have an f1.

Parsers generating these streams exist for highly structured data such as SGML (ISO8879:1986) or XML (Bray et al., 1988). SP (Clark) generates a field stream from an SGML document and a DTD. Other formats such as ASN.1 (ISO8824:1987) can also be converted into this format.

Both structured and unstructured documents can be parsed into the same stream. Unstructured documents have no field tokens.

### *3.2. The Document Tree*

The field stream is a walk through a tree, the document tree. The tree is initialized with a root when presented with the first document, and a pointer set pointing to it. When a field start token is encountered the pointer is adjusted to point to the appropriate child of the current node. If such a node is not found, it is created, and the pointer adjusted. When a field close token is encountered, the pointer is moved up in the tree. At the end of each document the pointer should have returned to the root.

During construction each node is given a unique identifier (the field id), starting with zero for the root and incrementing by one for each new node created. A walk of unstructured data will result in a tree with a root node having the field id zero and no children.

Once the field stream is walked for all documents, the tree represents every single path in the corpus, and each node has a unique field identifier. The tree is unlikely to be an exact match of paths from any one document, but it does match all paths from all documents. This tree, like that produced by Kotsakis (2002), is extensible. Once indexing is complete new documents can be added, even if they are in a new structure (perhaps the DTD has changed over time).

Terms can occur at any node or any leaf of the tree. Searching a document for the existence of a term within a given tag therefore requires searching at and below tree nodes.

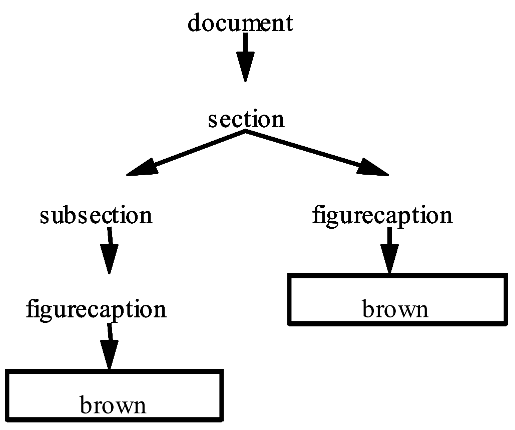

Figure 2: “figurecaption” is multiple and disjoint

Nodes are not atomic search units for some queries. In the structure represented in figure 2, “figurecaption” lies at two disjoint locations in the tree. A search for “brown in figurecaption” requires a search across multiple disjoint nodes. Lee et al. (1996) suggest searching at the parent node “section”, however this would incorrectly find “brown in subsection”. Some queries must be resolved in many locations, including both leaves and nodes of the tree. Shin et al. (1998) use a separate inverted file for each node and walk the tree looking for occurrences, however the loading multiple inverted lists is inefficient.

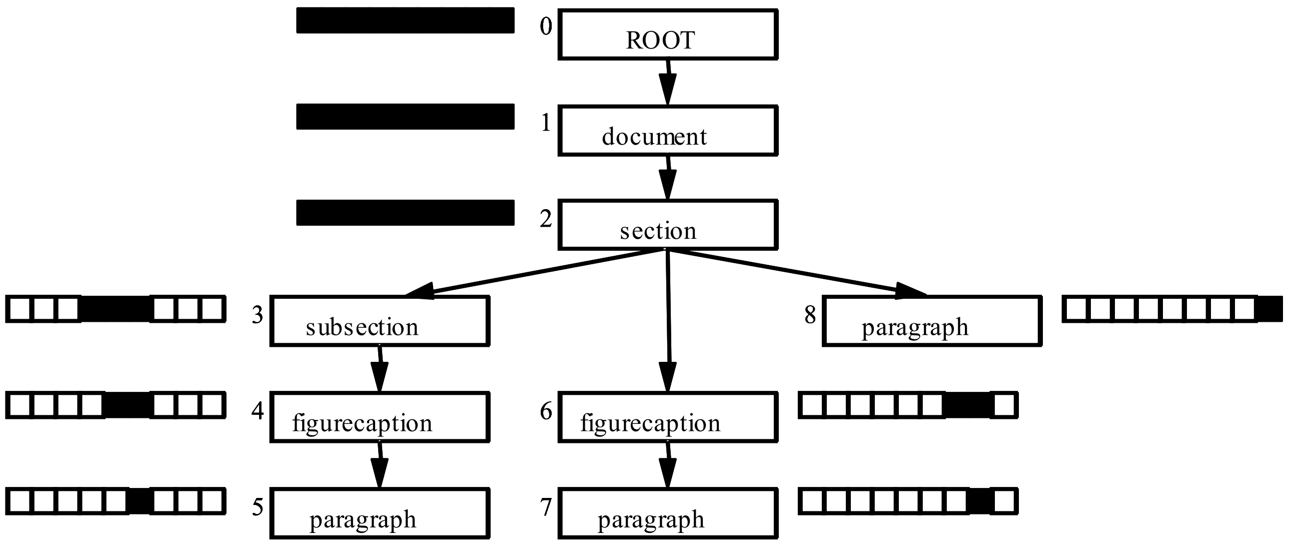

Figure 3a: Bit-strings stores at nodes

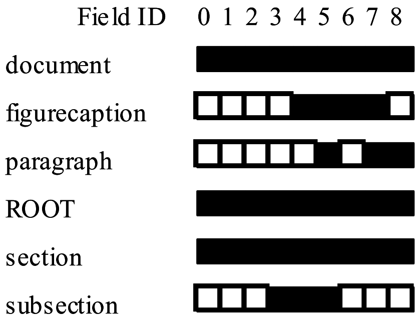

Figure 3b: The tree is flattened

The many to one relationship between tree nodes and tag name is stored in a list. Each time the tree is altered (before the first search) the tree is walked from leaves to root constructing at each node a list of field ids at and beneath that node. The lists are then combined to generate a list of ids for each tag. These lists are encoded in a bitstring (one bit per id) of length "nodes in tree". Creating these lists is done through a series of bitwise OR operations. An example of a tree and encoding is given in figure 3.

The field list represents every single unique tag in the collection alongside a list of the ids at and beneath (within) where it lies in the document tree. If a tag occurs multiple times in disjoint parts of the tree, this will be reflected in the field list.

Content searching structured documents can now be mapped to constructing a set of nodes from the document tree. To search for a given term anywhere in the document, the set is the full set. To search for a given term in a given tag, the set is taken from a binary search of the field-list. To search for a fully specified path, that path is followed top-down through the document tree. To search for a partially specified path, a bottom-up search of the document tree is done starting at the nodes identified by the field list.

## *3.3. Indices*

As each term in the field stream is parsed, it is given three attributes that collectively form a posting:

**record id** – the record's ordinal number; starting from zero and incrementing by one for each end record token.

**field id** – the current location in the field tree.

**position id** – the term's ordinal number, irrespective of field or record; starting at zero and incrementing with each term-token encountered. It is not incremented for field tokens or record tokens.

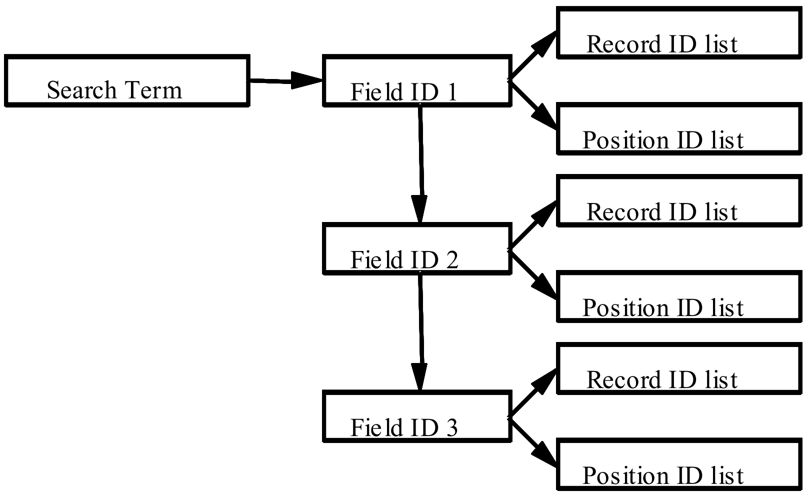

Figure 4: In-memory structure

For each unique term, these ids are collected together in the structure shown in figure 4. First the postings are grouped by field id then the record ids are collected together and the position ids are collected together. The field ids are stored in a sorted unique list alongside pointers to the record and position postings for that field.

Should a term occur multiple times in a single document it will occur multiple times in the postings. Should a term occur exactly twice in a single document, it will be represented exactly twice in the postings, irrespective of where in the document structure it lies.

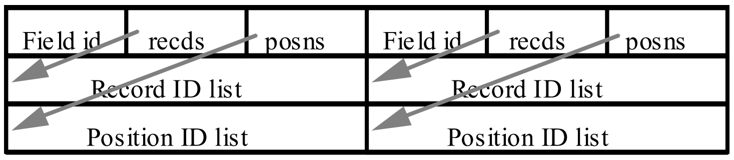

Figure 5: On-disk structure

When written to disk, the postings structure is converted into the linear structure shown in figure 5. The record ids are stored before the position ids to avoid unnecessary loading of position ids when not required as part of a search. Position ids are only required for a phrase search.

## 4. Searching and relevance

### *4.1. The Search*

One vocabulary file is used and is stored as a B-tree. Each vocabulary term has one postings list. Retrieval of the postings is done using techniques described by Zobel, Moffat, and Ramamohanarao (1998) and others (Salton & McGill, 1983).

Structured queries are explicitly marked with where to search. The query is parsed into a parse tree. The nodes of the tree represent Boolean operations, the leaves searchable terms (and where to search).

Where to search is converted from the symbolic representation of the query into a bitstring by walking the document tree and the field list as described above. This bitstring is a representation of the set of document tree nodes in which to search.

The vocabulary file is a unique list of all searchable terms in the corpus. Each entry in the vocabulary file has three attributes: the term, the file location of the postings for that term, and the length of the postings for that term. Given a search term from a query, a walk of the B-tree returns the location and length of the postings, a disk read from that location loads the postings into memory.

The retrieved postings may contain postings for unwanted fields. The set of where to look has been generated from the query, but may represent fields not in the postings. Selection of the required postings is done by walking the postings field-id-list. Each member of the list is checked for membership in the required set (a bit test). If it exists the postings are used. If not they are not. Discarded postings are not decompressed and are not used for searching.

The techniques of Meuss and Strohmaier (1999) and of Burkowski (1992) involve examining each posting and determining if it is appropriately located or not. This is inefficient. When postings are grouped by field id, it is possible to discard multiple postings in one comparison.

### *4.2. The Relevance*

Relevance calculations are performed using TF.IDF (Harman, 1992).

$$IDF_i = \log_2 \frac{N - t_i + 1}{t_i} \tag{1}$$

where N is the number of documents in the collection, and $t_i$ is the number of occurrences of term $i$ in the collection.

$$TF_{ij} = \frac{t_{ij}}{T_j} \tag{2}$$

or

$$TF_{ij} = \sum TF_{ijf} \tag{3}$$

$$TF_{ijf} = \frac{t_{ijf}}{T_j} \qquad (4)$$

where $t_{ijf}$ is the number of occurrences of term $i$ in field $f$ of document $j$, and $T_j$ is the length in terms of document $j$.

The influence of each term with respect to each document ( $w_{ij}$ ) is given by:

$$w_{ij} = TF_{ij} \times IDF_i \qquad (5)$$

These weights are summed for each search term to determine the relevance of a document with respect to a given query.

Kotsakis (2002) suggests assigning weights to each field. Replacing the desired field set from the query with a weighting array (zero weight to exclude a field) gives the same construction. The new relevance calculation becomes:

$$w_{ij} = \sum (TF_{ijf} \times C_f \times IDF_i) \qquad (6)$$

where $TF_{ijf}$ is the frequency of term $i$ in field $f$ of document $j$, and $C_f$ is a weighting constant for field $f$.

By extension, the user's query can be tagged with these field weights.

### *4.3. Phrase Searching*

A phrase may be distributed across multiple field tree nodes. Take for example the following XML:

<life><genus>Escherichia</genus><species>coli</species></life>

A search for the phrase "'Escherichia coli' in life" finds Escherichia in "genus in life", and coli in "species in life". From the field list, both genus and species are known to be children of life, so a search for "Escherichia coli in life" will find both terms. They will not, however, be found adjacent because each field is examined in turn.

For phrase searching, postings must be examined sequentially. The postings are loaded from disk, unwanted postings discarded, and the remaining postings merged. A phrase is recognized when all terms are in the same record and the position ids are sequential and in the right order. While indexing, the position id is only incremented when a searchable term is encountered. Consequently field tokens do not create gaps in phrases.

## 5. Analysis

### *5.1. Indexing*

The document tree is built directly from the field stream. Each time a new field start token is encountered, a new node must be built in the document tree. A search at the "current node" is required to determine if the field has been encountered before. If so no work is necessary. If not a new node must be created. The search can be implemented using a hash table.

The worst case for constructing the tree occurs when the document is "flat structured". Such documents occur when relational database tables are exported in XML. In this case, if there are *F* unique fields, and *f* total field occurrences, the cost of constructing this document tree is *O(f).* Each occurrence in the corpus must be examined once; each requires one hashed lookup of all the unique fields previously examined at the current node.

Traditionally a posting is represented by two ids (the record id and the position id). The proposed method adds a third, the field id. The structure build in memory during indexing is shown in figure 4. The worst case occurs when a term occurs in every unique field. Such might be expected from a term that occurs with a very high frequency such as "the". The cost of adding a new posting is *O(1)*. Each addition requires one array index to determine which postings list to use and one append for the posting itself.

## *5.2. Searching*

Query response time depends on disk I/O and network bandwidth (Moffat & Zobel, 1996). As the posting size increases, the storage space on disk and thus load time increases. Compression is often used to reduce index size and thus increase response time. Compression has not been examined in this investigation as it is an independent issue.

In an unstructured information retrieval system, once the posting is loaded, each posting must be examined at most once. Searching is $O(p)$ where $p$ is the number of postings for a given term.

The worst case for the proposed method occurs when each occurrence occurs in a unique field, and all fields must be examined. In this case, there are $p$ fields and $p$ occurrences.

A "where to look" bitstring of length $p$ must be constructed from the query and each bit must be set (this is $O(p)$).. The postings are loaded from disk, and the field id list of length $p$ traversed. Each list has one posting; there are $p$ postings to examine. Each list is examined once, each posting is then examined once. The overall cost is $O(p)$, where $p$ is the number of postings for a given term. This is linear with respect to the number of postings. No order change has occurred against unstructured searching.

In real data, the number of unique fields $F$ is usually very small, and the number of documents $N$ very large. For the TREC WSJ collection $F=20$ and $N=173,252$. In this case, for a term occurring once in every field and only once every document, the cost is $O(F + N)$. As the size of the corpus increases, the impact of $F$ tends to insignificant.

## *5.3. Phrase Searching*

Determining adjacency of two terms is analogous to merging two lists. Each term in one list must be examined against terms from the other to determine whether or not they are adjacent. Merging is $O(n \log_2 k)$ where $n$ is the size of the merged list, and $k$ is the number of lists. For two lists, $k = 2$ and $\log_2 k = 1$. Two term adjacency searching is therefore $O(p+q)$ where $p$ is the length of the postings for one term and $q$ the length of the postings for the second term.

Using the proposed method, the worst case occurs when, for both terms, each occurrence is in a different field. For the first term, there are $p$ occurrences in $p$ fields. The occurrence lists are merged in $O(p \log_2 p)$. Likewise for the second term, the merge is, $O(p \log_2 p)$. The total cost of determininag adjacency for two terms is $O(p \log_2 p + q \log_2 q)$.

The worst case is very unlikely to occur. In real world data, the number of fields is usually very small, and the number of occurrences large. In this case the cost tends more towards $O(p+q)$; linear with respect to the number of occurrences that must be examined.

## *5.4. Structured Searching*

A significant gain in efficiency occurs when searching for a term in only part of a document. Once the postings are loaded, examining the list of field ids will identify the relevant postings. These are then processed sequentially. Examining the field id list costs $F$ operations, examining the reduced postings takes $p'$ operations. The overall performance is $O(F + p')$ or linear with respect to the number of relevant postings. Other methods, such as that of Thom *et al.* (1995), are linear with respect to the total number of postings.

## *5.5. Effectiveness*

When a search occurs across all fields it is necessary to examine all occurrences of a given term. Although they are ordered by field, and examined in field order, each occurrence is examined only once. Since the influence of a term on document weight is a factor of TF, the term frequency, the order each is examined does not matter (see equation 3). The computed document query weight is the same regardless of what order the documents, the terms, and the occurrences of those terms are examined. Documents weights and precision metrics are the same as those computed with systems not using structured indexes. The proposed method adds functionality with no loss of existing functionality, impact on precision or impact on recall.

## 6. Results

Implementation is in C++ and has been tested on the Cystic Fibrosis document collection (Shaw, Wood, Wood, & Tibbo, 1991). CF is a 6MB, 1239 XML-document subset of MEDLINE and comes with 99 judged queries.

Indexing took less than 10 seconds on a Pentium III personal computer. Performing all 99 queries took less than five seconds, averaging 0.05 seconds per search. With commonly stopped terms removed from the queries, the test took less than one second.

The same test was performed against an 11,240,415 document subset of MEDLINE (43GB data, 13.5GB postings). Commonly stopped terms were removed from the queries, but not the indexes. The time taken to perform all 99 queries was 115 seconds, averaging 1.16 seconds per search. This test was performed on a Compaq AlphaServer ES40, with shared networked disk.

Indexing the 173,252 document TREC Wall Street Journal collection (1987-1992) from the TREC collection disks 1 and 2 took under 7 minutes on a Pentium 4 computer. The source XML was 533 Megabytes; the postings were 380 Megabytes. Topics 100-150 were used to test searching. Queries were built by extracting the description field then stopping commonly used words. Topic 121 was dropped as it has less than 5 known relevant documents. It tool less than 9 seconds to perform all 49 searching, averaging 0.18 seconds per search.

## 7. Conclusions

Structured formats such as SGML, XML, and ASN.1 are becoming ubiquitous so methods for searching structured documents are becoming important. Determining how a user could search structured documents has been studied by others, the culmination of whose work is evident in XML query languages.

Study of existing structured query systems alongside analysis of query languages has shown a miss-match. Either the systems do not support the necessary constructs, or to do so they forfeit recall, or efficiency.

A cross corpus document tree is built on the fly so new documents can be added, even if they are in a structure never before seen. All possible structure restricted searches can be described as a search for a term limited to existing in a set of locations taken from this tree. This set is represented as a bitstring for fast lookup.

Query paths are often only partially specified. The field list is constructed as an index to the document tree. Partially specified paths are resolved by branching into the tree at locations from the list and walking leaf to root. Without the field list a top-down search of the entire tree would be required.

For each occurrence of each term in the corpus a posting is constructed from three attributes, the record in which the occurrence lies, the ordinal number of the term since the beginning of the corpus, and the location in the document tree in which the occurrence was found. The postings are clustered by location in document tree. By clustering this way, it is possible to discard postings outside the search domain without examination.

Searching for the presence of structure is likened to searching for content (Schlieder & Meuss, 2000). Structural elements are encoded as search terms and retrieved as if search terms. The leaves of the document tree are encoded as these terms. To identify the appropriate leaf terms the document tree is walked using the field list as an index.

Large-scale information retrieval systems often support ranking. Adaptation of the TF.IDF ranking scheme to incorporate document structure not only allows structural systems to ranks, but to weight individual terms based on where in the document they are found.

The presented method allows searching for terms in a document, or in part of a document. Searching for phrases in documents and phrases that lie across tag boundaries is also supported. The implementation demonstrates the scalability of the system.

## References

Beitzel, S. M., Jensen, E. C., Grossman, D. A., Ingham, F. J., & Lewis, T. (2001). Using a relational database management system to implement XML-QL. In *Proceedings of the 17th International Conference on Advanced Science and Technology (ICAST'2001)*.

Bray, T., Paoli, J., & Sperberg-McQueen, C. (1988). Extensible markup language (XML) 1.0, W3C recommendation. Available: http://www.w3.org/TR/REC-xml.

Burkowski, F. J. (1992). Retrieval activities in a database consisting of heterogeneous collections of structured text. In *Proceedings of the 15th ACM SIGIR Conference on Information Retrieval*, (pp. 112-125).

Chinenyanga, T. T., & Kushmerick, N. (2001). Expressive retrieval from XML documents. In *Proceedings of the 24th ACM SIGIR Conference on Information Retrieval*, (pp. 163-171).

Clark, J. SP. Available: http://www.jclark.com/sp/.

Dao, T. (1998). An indexing model for structured documents to support queries on content, structure and attributes. In *Proceedings of the Advances in Digital Libraries*, (pp. 88-97).

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., & Suciu, D. (1998). XML-QL: A query language for XML. Available: http://www.w3.org/TR/NOTE-xml-ql/.

Fuhr, N., & Gövert, N. (2002). Index compression vs. Retrieval time of inverted files for XML documents. In *Proceedings of the 11th ACM International Conference on Information and Knowledge Management*.

Fuhr, N., & Großjohann, K. (2001). XIRQL: A query language for information retrieval in XML documents. In *Proceedings of the 24th ACM SIGIR Conference on Information Retrieval*, (pp. 172-180).

Harman, D. (1992). Ranking algorithms. In W. B. Frakes & R. Baeza-Yates (Eds.), *Information retrieval: Data structures and algorithms* (pp. 363-392). Englewood Cliffs, New Jersey, USA: Prentice Hall.

ISO8824:1987. (1987). *Information processing - open systems interconnection - specification of abstract syntax notation one (ASN.1)*.

ISO8879:1986. (1986). *Information processing - text and office systems - standard generalised markup language (SGML)*.

Kotsakis, E. (2002). Structured information retrieval in XML documents. In *Proceedings of the ACM Symposium on Applied Computing*, (pp. 663-667).

Lee, Y. K., Yoo, S.-J., Yoon, K., & Berra, P. B. (1996). Index structures for structured documents. In *Proceedings of the 1st ACM International Conference on Digital Libraries*, (pp. 91-99).

Meuss, H., & Strohmaier, C. (1999). Improving index structures for structured document retrieval. In *Proceedings of the 21st Annual Colloquium on IR Research (IRSG'99)*.

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

Robie, J., Lapp, J., & Schach, D. (1998). XML query language (XQL). Available: http://www.w3.org/TandS/QL/QL98/pp/xql.html.

Salton, G., & McGill, M. J. (1983). *Introduction to modern information retrieval*: McGraw-Hill.

Schlieder, T., & Meuss, H. (2000). Result ranking for structured queries against XML documents. In *Proceedings of the DELOS Workshop on Information Seeking, Searching and Querying in Digital Libraries*.

Schlieder, T., & Meuss, H. (2002). Querying and ranking XML documents. *Journal of the American Society for Information Science and Technology,* 53(6), 489-503.

Shaw, W. M., Wood, J. B., Wood, R. E., & Tibbo, H. R. (1991). The cystic fibrosis database: Content and research opportunities. *Library and Information Science Research,* 13, 347-366.

Shimura, T., Yoshikawa, M., & Uemura, S. (1999). Storage and retrieval of XML documents using object-relational databases. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, (pp. 206-217).

Shin, D., Jang, H., & Jin, H. (1998). BUS: An effective indexing and retrieval scheme in structured documents. In *Proceedings of the 3rd ACM International Conference on Digital libraries*, (pp. 235-243).

Thom, J. A., Zobel, J., & Grima, B. (1995). *Design of indexes for structured documents* (CITRI/TR-95-8). Melbourne, Australia: Department of Computer Science, RMIT.

Zobel, J., Moffat, A., & Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *Transactions on Database Systems,* 23(4), 453-490.