Indexing Documents for Queries on Structure, Content and Attributes

Ron Sacks-Davis Tuong Dao James A. Thom Justin Zobel

Multimedia Database Systems
RMIT, GPO Box 2476V, Melbourne 3001
Australia
{rsd,tuong}@mds.rmit.edu.au, {jat,jz}@cs.rmit.edu.au

Abstract

Indexing and retrieval techniques for large text databases are well developed, but most of the techniques developed to date assume that the text to be indexed has little or no structure. With the growth in the use of sophisticated markup languages for text, a database system for structured documents should use, not just document content, but structural information and attributes, and should support queries on content, structure and attributes. In this paper we review and compare two recent approaches for accessing document collections. For one of the approaches, position-based indexing, queries are resolved by manipulating ranges of word offsets while for the other, based on a path model, the position of a word is represented in terms of the structural components that enclose it. The former allows slightly smaller indexes; the latter allows more efficient query evaluation.

1 Introduction

Techniques for indexing large text databases are well developed; and multigigabyte text collections can be accessed with sub-second response time [9]. However, most of the techniques developed to date have been designed for text with little or no structure.

Documents consist of components which conform to one or more hierarchical structures. For example, the logical structure of a textbook might be a hierarchy consisting of chapters containing sections containing paragraphs, whereas the layout structure may consist of pages containing lines. Documents, and document components at all levels of a document hierarchy, can have attributes. Depending on the application attributes can represent a wide range of information about a document or document component, such as who wrote it, who can access it, what language it is written in, and what version it is.

A database system for structured documents should capture, not only the content of documents, but also the structural information and the attributes of documents and document components, and should support queries on content, structure and attributes. Such structure can be recursive—lists can contain lists, for example. Also, documents can vary widely in structure; two documents within the same collection can have completely different subcomponents and widely varying size and complexity. Moreover, although many documents have a single hierarchical logical structure, document representation standards such as SGML [7] allow for the creation of documents with multiple hierarchies. These considerations make it difficult to design effective indexes for large databases of structured documents.

The needs of document database systems are not met by the indexing techniques designed for the standard database applications [13]. Traditional relational database management systems require strict table-oriented data, and are based on the assumption that a schema is defined in advance and adhered to by all data managed by the system. While object-oriented database systems permit much richer structure than relational database systems, they still require that data conform to a predefined though more flexible schema. On the other hand, the structure of document collections can evolve rapidly. SGML collections require that the structure be predefined, but widely differing document structures can be supported within a single collection. For XML collections there is no requirement for a document schema. As a consequence, the indexing methods used for document collections cannot necessarily assume that the data to be stored has any fixed or regular structure.

In this paper we describe techniques for accessing document collections by content, structure, or attributes. Recent models of text databases have attempted to integrate both contents and structure in text retrieval (see for example the surveys by Baeza-Yates and Navarro [1, 11]). However none of these models allow for querying of attributes at arbitrary levels of the document hierarchy. We present and compare two approaches for indexing document collections, path-based indexing and position-based indexing. We conclude that both methods provide robust mechanisms for effectively supporting queries on content, structure and attributes but, while path indexes may be slightly more costly of storage, they are more efficient for many commonly occurring query types.

2 Structured Documents

An SGML document consists of logical components called *elements*. For example a poem might contain elements such as <title>, <author>, <stanza>, and <l>ine. A DTD (document type definition) is a grammar that specifies the sub-elements and the matter that each element can contain; a typical definition might specify that a <stanza> consists of one of more <l>ines, for example. Elements can be nested within each other, allowing recursion. In typical use of SGML, in a document that conforms to the DTD each element is surrounded by a begin tag and an end tag.

In addition, type information can be associated with elements by using attributes that appear in the text within the begin tag. Attributes contain metadata about the contents of documents and their components. For example, associated with the <stanza> element might be an attribute n that specifies the number of the stanza within the poem. Since attributes contain important information about a document, as well as being able to query on content and structure it is also valuable to be able to query on attributes. Sometimes such attributes may be implied rather than stored.

As an example consider the poem *Jabberwocky* by Lewis Carroll (see Figure 1). The poem consists of seven stanzas, indicated by the tags <stanza> and </stanza>; each stanza has four lines of verse, indicated by the tags <l> and </l>; there are words in speeches, indicated by the tags <q> and </q>; and proper names, indicated by tags <name> and </name>. Some elements of the hierarchy can have attributes; for example, <stanza> can have an attribute n, in-

```
    <doc>
    <front>
  1 <title>Jabberwocky</title>
    <author>
    <name type=person realname="Rev. Charles Dodgson">
  2 Lewis Carroll</name>
    </author>
    </front>
    <body>
    <stanza n = 1>
  4 <l>'Twas brillig, and the slithy toves </l>
 10 <l>Did gyre and gimble in the wabe:</l>
 17 <l>All mimsy were the borogoves,</l>
 22 <l>And the mome raths outgrabe.</l>
    </stanza>
    <stanza n = 2>
 27 <l><q>Beware the
 29 <name type = creature>Jabberwock</name>,
 30 my son!</l>
 32 <l>The jaws that bite, the claws that catch! </l>
 40 <l>Beware the <name type = creature>Jubjub</name>
 43 bird, and shun</l>
 46 <l>The frumious
 48 <name type = creature>Bandersnatch</name>!</q></l>
    </stanza>
    <stanza n = 3>
 49 <l>He took his vorpal sword in hand:</l>
 56 <l>Long time the manxome foe he sought&dash;</l>
 63 <l>So rested he by the <name>Tumtum</name>
 69 tree,</l>
 70 <l>And stood a while in thought.</l>
    </stanza>
    <stanza n = 4>
 76 <l>And, as in uffish thought he stood,</l>
 83 <l>The <name type = creature>Jabberwock</name>,
 85 with eyes of flame,</l>
 89 <l>Came whiffling through the tulgey wood,</l>
 95 <l>And burbled as it came!</l>
    </stanza>
    <stanza n = 5>
100  <l>One, two! One, two! And through and through</l>
108 <l>The vorpal blade went snicker-snack!</l>
113 <l>He left it dead, and with its head </l>
121 <l>He went galumphing back.</l>
    </stanza>
    <stanza n = 6>
125 <l><q>And, hast thou slain the
130 <name type = creature>Jabberwock</name>?</l>
131 <l>Come to my arms, my beamish boy!</l>
138 <l>Oh frabjous day! Callooh! Callay!</q></l>
143 <l>He chortled in his joy.</l>
    </stanza>
    <stanza n = 7>
148 <l>'Twas brillig, and the slithy toves </l>
154 <l>Did gyre and gimble in the wabe:</l>
161 <l>All mimsy were the borogoves,</l>
166 <l>And the mome raths outgrabe.</l>
    </stanza>
    </body>
    </doc>
```

Figure 1: *Jabberwocky* — a small sample document

dicating the number of the stanza, and `<name>` can have attributes `type`, and `realname`. Word offsets are shown before lines which contain words.

SGML itself is a meta-language; it describes how to form a DTD and how to parse DTD-conformant documents. Each DTD describes a particular language for marking up documents. The best-known SGML language is almost certainly HTML, which has become the language of the web. (However, web pages that are HTML-conformant are rare—most use the tags as defined in the HTML DTD but do not use them correctly.) Another important language is XML, which is in effect a simplified form of SGML. In XML the more complex aspects of SGML, such as tag omission, have been discarded, and DTDs and DTD-conformance are not mandatory.

3 Queries on Structured Documents

In this section we consider typical kinds of queries on collections of structured documents and the kinds of structures that are required for efficient evaluation of these queries.

3.1 Content

There are two major query paradigms for text, Boolean retrieval and ranked retrieval. Both are based on identifying the documents that contain the query terms; Boolean queries require an exact match while ranked queries use approximate or fuzzy match, ordering the documents, roughly speaking, according to the density of occurrence of the query terms.

Boolean queries, such as the following query, can incorporate the notion of adjacency or proximity, in which the query terms must be within a certain distance of each other in matching documents.

Boolean query

> *Query 1: Find documents whose author is Lewis Carroll and which contain the words "gyre" and "gimble" within 4 words of each other.*

As a result, for efficient processing of Boolean queries it is not sufficient for an index to just store the identifiers of documents containing each query term; the index must also store the positions at which each term occurs in each document.

In ranked retrieval, a query is expressed as a set of keywords or in natural language (and may even be an example document that was considered to be relevant).

Ranked query

> *Query 2: Rank documents about animals, birds, and beasts.*

The database system retrieves documents from the database to be displayed in order of their estimated relevance to the query [15]. Many functions for computing similarity have been devised; perhaps the most widely used is the cosine measure. These functions compute the similarity of a document to a query by assigning weights to each query term and to each term in a document. These term weights are computed during query evaluation and are based on statistics such as the frequency of a term in the database and the frequency of a term in a document. Similarity measures also require an estimate of the length of each document, usually measured as a function of the weights of the terms occurring in the document. As a result, for efficient processing the index must maintain statistical information about each term in the document collection, and the document lengths must be explicitly computed and stored as documents are inserted into the collection.

3.2 Structure

Queries on document databases can require that certain relationships hold between document elements and can impose constraints on the content of these elements. In addition, the unit of retrieval can be any document component as well as whole documents. In the following query, the unit of retrieval is the `<stanza>`s that contain a `<q>`uote with the word *Jabberwocky*.

Query on content and structure

> *Query 3: Find stanzas containing a quote that includes the word "Jabberwock"*

Ranked queries on content and structure

> *Query 4: Rank titles about animals, birds, and beasts.*

> *Query 5: Rank stanzas about Jabberwocks and Jubjubs.*

Although many documents conform to a single hierarchical schema, multiple hierarchies with overlapping structures are possible. Models that assume a single document hierarchy are not able to accommodate these documents.

Query involving multiple hierarchies

> *Query 6: Find quotes that are not contained in a line.*

Some queries may be concerned with structure only.

Query on structure

> *Query 7: Find documents that contain more that 5 stanzas.*

For efficient evaluation of these queries, information about the structure of documents must be maintained within the index. It must be possible to deduce the relationship of each word to the structure of the document and to compute relationships between the structural elements of a document. Because document schemas can evolve with time, the index structures must be able to accommodate new structural elements and new relationships between existing elements.

3.3 Attributes

Attributes are associated with the structural elements of a document, it follows that indexes for attributes must associate attribute values with elements rather than words.

Queries on attributes can include the associated element name, the name of attribute and the value of the attribute. However, sometimes either the element name or the attribute name may not be included in the query, indeed only the attribute value may be be given. The index must accommodate all of these possibilities. The following examples illustrate queries on attributes.

Query on attributes

Query 8: Find names with attribute `realname` = *"Rev. Charles Dodgson".*

Query on content, structure and attributes

Query 9: Find lines in stanza numbered 2 that include the word "Jabberwock".

Queries on attribute value or name

Query 10: Find elements with an attribute `type` = *"creature".*

Query 11: Find elements with an attribute `type`.

Query 12: Find elements with an attribute with value "creature".

Query 13: Find elements with an attribute "realname" and another attribute with value "person".

It is always possible to evaluate queries without an index, or to use an index to identify possible answers and then post-process to eliminate false matches, but such approaches are usually more time-consuming than fully supporting query evaluation in the index. In a collection of complex documents, fetching and parsing a single false match can be as costly as a great deal of index processing. If an index is to be used for full evaluation of the queries listed above, it must: store the identifiers of each document containing each query term, and the positions in these documents at which the term occurs; allow determination of which elements contain which terms; indicate which documents contain which structures; and associate attributes and attribute values with structures and terms. Two possible approaches to indexing are position-based indexes and path-based indexes; we review both of these in the following sections.

4 Position-Based Indexing

In position-based indexing, queries are resolved by manipulating ranges of offsets. The GCL (generalised concordance lists) position-based model and query language were proposed by Clarke et al. [4]. GCL is based on a data structure called a concordance list, which consists of text intervals called extents. Each extent is described by a start position and length (or equivalently end position). An extent could for example correspond to the word positions covered by a structural element such as a title or a line, or it could simply correspond to an arbitrary contiguous sequence of words. The GCL query language uses containment relationships, such as *containing* and *contained in*, between text intervals, to allow queries on document structure.

Each word occurrence is assigned an integer number corresponding to its position (offset) within the document; tags are ignored when computing offsets. Each tag is assigned a pair of numbers, the first indicating the position number of the preceding word and the second indicating the position of the tag in the current sequence of tags. The reason tags are not assigned an offset is so that they do not interfere with proximity searches on words. Considering the example document in Figure 1, the first word is the title "Jabberwocky" which would have position 1, the second word "Lewis" would have position 2, and so on. The first tag `<doc>` would have position 0.1, the second tag would have position 0.2, the third tag 0.3, the fourth tag `</title>` 1.1 as it come after the first actual word, the fifth tag 1.2, and so on.

GCL evolved from two earlier structured text retrieval languages developed at the University of Waterloo, the Pat text searching system [14], which was developed for use with the New Oxford English Dictionary, and a language described by Burkowski [2]. The extents in Burkowski's concordance lists are restricted to be disjoint, whereas in GCL overlapping extents are supported. However, GCL does not support the nesting of extents within each other (for example `<lists>` within `<lists>`). The GCL model was extended by Dao et al. [5, 6]to allow nested extents as well as overlapping extents. The resulting SCL (simple concordance lists) model supports the recursive structures present in many documents.

In the original implementation framework for GCL proposed by Clarke et al. [4], all terms in the database (both words and markup symbols) were indexed using an inverted file. Corresponding to each distinct term in the *lexicon*, an *inverted list* would be maintained, storing the document identifiers and offsets of each occurrence of the term. The advantage of this approach is that standard indexes for text databases could be used to implement the model. However, to produce the list of all occurrences of a structural element, such as a `<title>`, the inverted list corresponding to the open tag `<title>` would need to be retrieved as well as the inverted list for the close tag `</title>`.

The SCL model was proposed by Dao et al. [5, 6] in order to overcome the restriction of not allowing recursive structures. In SCL two index functions are supported: a *word index* that gives the document and the word position for each word in the document collection and an *element index* that gives for each structural element in the database, the document and positions of the start and end tags for each occurrence of that element. Thus both words and extents are indexed directly. The provision of an extent index also makes processing of queries more efficient, as extents do not need to be materialised by merging lists of open tag and close tag positions.

The SCL model uses a single data type, the SC-list. An SC-list is an ordered set of extents of the form (p, q) where p is the start position, q is the end position, and $p < q$. Extents in the SC-list are in ascending order with respect to their start positions and extents with the same start position are in descending order with respect to their end positions. Both nesting and overlapping are allowed in an SC-list. SCL database query operators operate on SC-lists and return SC-lists. To allow more efficient index processing, extents are not allowed to bridge documents; that is, documents are treated as distinct, independent entities.

For example, the query to return to all `<stanza>`s would return the following partial SC-list. (Here we have assumed that that *Jabberwocky* is the fifth document in the database.)

```
5:(3.5, 26.2)
5:(26.3, 48.4)
5:(48.5, 75.2)
5:(75.3, 99.2)
5:(99.3, 124.2)
5:(124.3, 147.2)
5:(147.3, 170.2)
```

where the first entry specifies that the first occurrence of a `<stanza>` in the fifth document starts at position 3.5 and ends at 26.2 and so on. The query to return all `<q>`uotes would return the set of extents

```
5:(26.5, 48.2)
5:(124.5, 142.1)
```

By intersecting these lists of extents it is possible to determine which `<stanza>`s contain `<quotes>`s. Similarly, by combining these SC-lists with the inverted list for the word *Jabberwock*:

```
5:29
5:84
5:130
```

it is possible to determine the result for Query 3, the SC-list representing `<stanza>`s containing a `<q>`uote that includes the word *Jabberwock*:

```
5:(26.3, 48.4)
5:(124.3, 147.2)
```

The components of a position-based indexing scheme are: a text index, consisting of a lexicon of terms and an inverted list for each term; and an element index, consisting of a lexicon of elements and an inverted lists for each element. These are required to support queries on content and structure. To support queries on attributes, an attribute index is also required. Attribute indexes are discussed in Section 6.

Each inverted list in the text index corresponds to a word in the database. Since there may be many occurrences of the same word within a document, it is not efficient to repeat the document number for each of these occurrences as indicated in the example above. Rather, the inverted lists hold sequences of monotonically increasing word positions in the format shown below.

$$\{< docno >< numpos > \{< pos > \ldots\} \ldots\}$$

The word positions are preceded by the corresponding document identifier and the number of positions for that document. Efficient variable-length representations of integers, and techniques such as taking of differences, can be used to reduce the space required to store such inverted lists; these are explained in Section 5.

Inverted file entries can be long. In order to avoid having to process a complete inverted file entry when only a small part of it is required, a map of each inverted list can be stored in the text lexicon. The map allows blocks of entries in the inverted list to be skipped. This index map is similar to that described by Clarke et al. [3], but applied to each inverted list rather than to the entire inverted file. It is similar in principle to the skipping scheme of Moffat and Zobel [10]. In typical cases in practice, most inverted lists must still be fetched in full, but only part of each list need be processed.

Each element inverted list corresponds to an element name in the database, is stored in one or more contiguous blocks, and holds a sequence of position pairs $< p, q >$in the format shown below.

$$\{< docno >< numpospairs > \{< p, q > \ldots\} \ldots\}$$

Each position pair represents the start position p and the end position q of a text interval occupied by a structural element. The position pairs are sorted in ascending order with respect to start position, pairs with the same start position are sorted in descending order with respect to end position. This allows the efficient processing of queries involving both structure and content, including queries on recursive document structures.

5 Path-Based Indexing

In traditional inverted list indexing, and in position-based indexing, word positions are represented as absolute word offsets from the start of the document. An alternative is the path model, in which the position of a word is represented in terms of the structural components that enclose it. In other words, the structure of documents is encoded in the index.

In order to determine the position of a word within a document, it is first necessary construct an encoding of the path of the element names from the root of the document to the leaf node containing the word, with the sibling number for each element. For each word occurrence, then, the inverted list includes a representation of the path to that word incorporating both elements and sibling numbers. A typical path representation is

```
doc 5, body 1, stanza 8, l 1, 7
```

representing that the indexed word is the seventh in the first line of the eighth stanza in the body of the fifth document.

Most queries involving structure and content can be evaluated efficiently using a path representation. For example, to answer query 3 involves retrieving the inverted list for the word *Jabberwock*:

```
doc 5, body 1, stanza 2, l 1, name 1, 1
doc 5, body 1, stanza 4, l 2, name 1, 1
doc 5, body 1, stanza 6, l 1, name 1, 1
doc 5, body 1, stanza 2, q 1, name 1, 1
doc 5, body 1, stanza 6, q 1, name 1, 1
```

from which it is evident that the fifth document contains two stanzas that satisfy the query. Compared to the position-based indexing scheme, no intersection of inverted lists is required for this query.

Note that, as the database structure contains overlapping elements, some redundancy is introduced into the index structures. For example, in the inverted list for the word *Jabberwock* occurrences of *Jabberwock* that occur in both lines and overlapping quotes are represented by two inverted file entries.

For the *Jabberwocky* document, the inverted list for the word *and* would include the following entries.

```
doc 5, body 1, stanza 1, l 1, 3
doc 5, body 1, stanza 1, l 2, 3
doc 5, body 1, stanza 1, l 4, 1
doc 5, body 1, stanza 2, l 3, 5
doc 5, body 1, stanza 3, l 4, 1
doc 5, body 1, stanza 4, l 1, 1
doc 5, body 1, stanza 5, l 1, 5
doc 5, body 1, stanza 5, l 1, 7
doc 5, body 1, stanza 5, l 3, 5
doc 5, body 1, stanza 6, l 1, 1
doc 5, body 1, stanza 7, l 1, 3
doc 5, body 1, stanza 7, l 2, 3
doc 5, body 1, stanza 7, l 4, 1
doc 5, body 1, stanza 2, q 1, 18
doc 5, body 1, stanza 6, q 1, 1
```

As discussed above, for queries involving the structure of documents, we can use the path information stored in the inverted index to determine whether a particular element contains the term by simply checking to see if the element name appears in the path. The encoding of the paths can be chosen to optimise the efficiency of this type of checking, and to reduce the space requirements—stored naively, such inverted lists would be prohibitively large. For example, one possibility is the encode the paths and the sibling numbers separately. The information may then be represented more compactly by not repeating the same path name for each occurrence of the word, as shown below.

```
doc, body, stanza, l
13
5, 1, 1, 1, 3
5, 1, 1, 2, 3
5, 1, 1, 4, 1
5, 1, 2, 3, 5
5, 1, 3, 4, 1
5, 1, 4, 1, 1
5, 1, 5, 1, 5
5, 1, 5, 1, 7
5, 1, 5, 3, 5
5, 1, 6, 1, 1
5, 1, 7, 1, 3
5, 1, 7, 2, 3
5, 1, 7, 4, 1
doc, body, stanza, q
2
5, 1, 2, 1, 18
5, 1, 6, 1, 1
```

Even for large document collections, the number of different paths to be represented will typically be small.

It is more efficient to store the differences between the sibling numbers (or word positions) than the sibling numbers themselves [10, 16]. The leading (left-hand) differences will be zero, and in general it is more efficient to store the number of zero differences than to store a string of zeros. The above entries can then be represented as follows. The number in brackets represents the number of fields (counting from the left) in the entry for which differences are not stored. For the next field, the integer represents relative values of sibling numbers or word positions. The remaining fields contain the absolute values of the sibling numbers or word positions.

```
doc, body, stanza, l
13
5, 1, 1, 1, 3
(3) 1, 3
(3) 2, 1
(2) 1, 3, 5
(2) 1, 4, 1
(2) 1, 1, 1
(2) 1, 1, 5
(4) 2
(3) 2, 5
(2) 1, 1, 1
(2) 1, 1, 3
(3) 1, 3
(3) 2, 1
doc, body, stanza, q
2
5, 1, 2, 1, 18
(2) 4, 1, 1
```

The integers representing the relative the sibling numbers can then be efficiently stored using a nonparameterized code such as the Elias gamma or delta codes, or with a parameterised Golomb code [16]. If such variable-length codes are used a little extra housekeeping information is required, to allow the index parser to distinguish paths from the other information. Similar techniques can be used to reduce the space required for position indexes.

There are many small variants on these schemes, such as front-coding of paths, that is, storing partial paths with a leading count to indicate how many elements in the current path occurrence are repeated from the previous path occurrence.

To answer queries involving structure only, it is necessary to maintain inverted lists for the structural elements. To represent elements using the path model, the complete path in the tree down to the corresponding element is used. For example, the first occurrence of the `<name>` element in the example document can be represented by the path

```
doc 5, front 1, author 1, name 1
```

Similarly, the inverted list for the `<stanza>` element will contain the following entries:

```
doc, body, stanza
7
5, 1, 1
5, 1, 2
5, 1, 3
5, 1, 4
5, 1, 5
5, 1, 6
5, 1, 7
```

and by retrieving this inverted list it is straightforward to determine that document 5 contains seven stanzas, thereby providing an answer to Query 7 on structure.

6 Indexing Attributes

Attributes are associated with the structural elements of a document, it follows that indexes for attributes must associate attributes and attribute values with extents rather than words.

The inverted lists for attributes and attribute values are subsets of the element inverted lists. Each such inverted list is associated with a <*element,attribute,value*> triple. Note that queries on attributes can specify the element name, the attribute name and the attribute value or any subset of these. For example, Query 8 specifies that the element name is `<name>`, the attribute is *realname* and the attribute value is *Rev. Charles Dodgson*. To answer this query it is necessary to retrieve a single inverted list. However for Query 10, only the attribute *type* and value *creature* is specified and there may be many qualifying elements with this combination of attribute name and attribute value. Similarly, for Query 11 only the attribute name is specified while for Query 12 only the attribute value is specified. To answer these latter three queries it is possible that several inverted lists must be retrieved.

A table of <*element,attribute,value*> triples must be maintained as part of the attribute index. In our example the table will contain the following entries:

element	attribute	value
name	type	person
name	type	creature
name	realname	Rev Charles Dodgson
stanza	n	1
stanza	n	2
stanza	n	3
stanza	n	4
stanza	n	5
stanza	n	6
stanza	n	7

There are many possible representations and data structures for this table. One option is to represent the table as an SGML (or XML) document, so that it can be queried using the same query language that is used for searching the document database. The data structures to be used for representing the table should allow for efficient identification of the matching triples when only a subset of the <*element,attribute,value*> information is provided.

Associated with each <*element,attribute,value*> triple is an inverted list of extents. Depending on the relative size of the inverted list and the corresponding element inverted list, the inverted list for each triple can be represented in several ways, including:

- *Element inverted list*: The inverted list for a <*element,attribute,value*> triple is identical in organisation to the element inverted list but is maintained separately from the corresponding element inverted list.
- *Bitmap inverted list*: The inverted list for the <*element,attribute,value*> triple is represented by a bit map with the same number of entries as there are in the corresponding element inverted list. The bit map identifies the entries in the element inverted list (that is, the extents) for which the value of the attribute applies.
- *Interval inverted list*: This inverted list for the <*element,attribute,value*> triple indicates the groups of consecutive extents in the element inverted list for which the attribute value applies.

For example, using the position-based index, the inverted list for the element `<name>`, stored in the element index, is

```
5:(1.3, 3.1)
5:(28.1, 29.1)
5:(41.1, 42.1)
5:(47.1, 48.1)
5:(67.1, 68.1)
5:(83.1, 84.1)
5:(129.1, 130.1)
```

and the inverted list for the <*name,type,creature*> triple, stored in the attribute index, is

```
5:(28.1, 29.1)
5:(41.1, 42.1)
5:(47.1, 48.1)
5:(83.1, 84.1)
5:(129.1, 130.1)
```

which might also be represented using a bit vector

```
[0,1,1,1,0,1,1].
```

For the path-based model, the inverted list would be represented as

```
doc body stanza l name
6
5 1 2 1 1
5 1 2 3 1
5 1 2 4 1
5 1 3 3 1
5 1 4 2 1
5 1 6 1 1
doc body stanza q name
4
5 1 2 1 1
5 1 2 1 2
5 1 2 1 3
5 1 6 1 1
```

where the additional entries are due to the fact that the extents for the element `<name>` are contained in overlapping `<l>`ine and `<q>`uote extents.

7 Path-Based Indexes versus Position-Based Indexes

Path indexes and position indexes each have distinct advantages and disadvantages. The two methods have different behaviour in the presence of updates. If a word is inserted towards the start of a document, then with the position-based indexing the positions of all the following words and structural elements change, requiring updates to a large number of inverted lists. For the path indexes, changes in word positions are confined to the corresponding leaf element of the document tree, so updates are less expensive.

On the other hand, the path indexes can only be used to computed the differences in word positions for two words located in the same leaf element of the document tree. This restriction can be overcome by recording the absolute word offsets from the start of the document in the path indexes rather than the offsets from the start of leaf element. With this modification, however, the path indexes will exhibit the same behaviour as the position-based indexes in the presence of updates.

Perhaps the best approach, for path indexes, is to identify some elements as too minor to justify introduction of a new context, and not restart the offset numbering when such an element is encountered. In our example, word numbering might, for instance, be reset at stanza boundaries but not line boundaries. Such an approach requires additional information about elements such as might be provided by a DTD; the difficulties of distinguishing between kinds of elements are a disadvantage of the use of dynamic schemas.

Additional markup tags introduced in documents to identify new structural elements can be easily accommodated in position indexes which simply require an additional inverted list for each new structural element. Similarly individual element inverted lists can be removed from position indexes if corresponding structural elements are no longer required in queries. In the case of path indexes, the addition or removal of a single structural level in the hierarchy potentially involves the reconstruction of the whole index.

Both methods provide efficient support for the types of queries introduced in section 3. However, because of the way the respective indexes are constructed, both path indexes and position indexes are unable to efficiently support some types of queries. Structural queries about the position of elements (such as "find `<q>`uotes that begin at the start of a `<l>`ine") cannot easily be evaluated with path indexes because absolute position information on elements is not maintained. Similarly, position indexes do not provide direct support for queries on the hierarchical relationship between elements (such as "find the third `<l>`ine of the second `<stanza>`"). For position indexes, the hierarchical structure must be deduced from the positional information about elements.

In principle, the two approaches have roughly similar storage requirements. Assuming a single hierarchy, both schemes allow exact reconstruction of the original documents and contain no unnecessary information for this task; and the entropies of the two representations are similar. However, in practice path indexes require somewhat more storage overhead, because the individual values to be coded are smaller and (with integral-bit codes that can be rapidly decoded) are not represented as efficiently. As a broad estimate, a position index requires around 30%–35% of the size of the indexed data and a path index requires around 40%–45% of the size of the indexed data.

The main advantage of path indexes is the efficiency of query evaluation. Consider example a query to retrieve the documents where a given element contains a particular word. With path indexes, a single inverted list is retrieved, whose length is determined by the frequency of the word. In the TREC data [8], for example, a typical query term occurs in around 0.1% of the stored documents and the inverted list would occupy a small number of disk blocks. For position based indexes, two inverted lists must be retrieved, one for the element and one for the word, and these lists must be intersected. The word inverted list will be somewhat smaller than the corresponding list in the path index, but, if the element occurs in most documents, then its inverted list will be long and the cost of retrieval of this list will dominate query evaluation costs. For the commonest elements, such as `<sentence>`s and `<l>`ines, the inverted lists will be very long indeed. Strategies such as skipping can ameliorate the cost of accessing these long inverted lists, but nonetheless these costs will dominate.

8 Ranking Queries

Traditional information retrieval systems do not make use of document structure, and support ranking of whole documents only. Such ranking is relatively easy to support: the index need only store the overall frequency of each term and, for each document containing each term, the in-document frequency. Document lengths can be held in a separate table, which is usually small enough to be held in memory. Word positions are not required.

Both path indexes and position indexes can be used for document-level ranking, although not as efficiently as indexes without positional information. In the presence of overlapping hierarchies, however, there are problems with path indexes; as the examples in the previous section show, if there are overlapping hierarchies the occurrence of a word in, say, a `stanza` can be recorded more than once. A solution is to explicitly store the true frequency of the word in the document. Overall, if document-level ranking queries are to be supported it may be preferable to explicitly create an additional index holding only the information required for ranking; such an index typically requires less than 10% of the space required for the indexed data, and can be specifically organised to allow rapid ranking with minimal disk traffic [12].

For ranking of arbitrary elements—that is, to allow ranking queries on any part of a document, not just whole documents—two problems must be solved: determination of in-element frequencies and of element lengths. If there is a single hierarchy, the path index can be used to compute in-element frequencies, but in in the general case of overlapping hierarchies this information is not accurately available and must be stored explicitly for every element occurrence, incurring a significant cost. In-element frequencies can be obtained for position indexes after intersecting the appropriate element and word inverted lists. For both the position indexes and the path indexes, the element lengths can be integrated into the existing element inverted lists, in the case of position indexes element lengths can be approximated from the number of word occurrences in each element.

In neither case, however, is ranking cheap. The sheer number of entities to rank means that large numbers of accumulators must be maintained, and long inverted lists processed in full. In a production system, rather than allowing ranking on arbitrary elements it may well be preferable to specify which elements can be ranked, allowing construction of special-purpose structures and use of more efficient ranking algorithms [10, 12].

9 Conclusions

Queries on text databases can involve structure, content and attributes. To support efficient processing of such queries, new index structures are required. In this paper we have reviewed two alternative approaches to building such indexes. One approach uses the position of words and elements for the index, the other uses paths through the document structure to index words.

The main advantage of the path indexes is the efficiency of query evaluation. Storage costs for the path indexes are greater than for the position indexes, but not significantly so if the types of compression techniques we have presented for path indexes are used.

Both methods can be used to effectively support queries on content, structure and attributes. Ranking queries on arbitrary elements will be expensive, though, whichever method is used.

Acknowledgements

We would like to thank Tim Arnold-Moore, Marcello Cantos, and Bruce Grima. This work was supported by the Australian Research Council.

References

[1] R. Baeza-Yates and G. Navarro. Integrating contents and structure in text retrieval. *ACM SIGMOD Record*, 25(1):67–79, March 1996.

[2] F. J. Burkowski. An algebra for hierarchically organized text-dominated databases. *Information Processing and Management*, 28(3):333–348, 1992.

[3] C. L. Clarke and G. V. Cormack. Dynamic inverted indexes for a distributed full-text retrieval system. Technical Report MT-95-01, University of Waterloo, 1994.

[4] C. L. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[5] T. Dao, R. Sacks-Davis, and J. A. Thom. Indexing structured text for queries on containment relationships. In *Proceedings of the 7th Australasian Database Conference*, pages 82–91, Melbourne, Australia, January 1996.

[6] T. Dao, R. Sacks-Davis, and J. A. Thom. An indexing scheme for structured documents and its implementation. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications*, pages 125–134, Melbourne, Australia, April 1997.

[7] C. F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.

[8] D. Harman. Overview of the second text retrieval conference (TREC-2). *Information Processing & Management*, 31(3):271–289, 1995.

[9] A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Computer Journal*, 26(1):1–9, 1994.

[10] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[11] G. Navarro and R. Baeza-Yates. Proximal nodes: a model to query document databases by contents and stucture. *ACM Transactions on Information Systems*, (to appear).

[12] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[13] R. Sacks-Davis, T. Arnold-Moore, and J. Zobel. Database systems for structured documents. *IEICE Transactions on Information and Systems*, 34-D(11), 1995.

[14] A. Salminen and F. W. Tompa. Pat expressions — An algebra for text search. *Acta Linguistica Hungarica*, 41:277–306, 1994.

[15] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[16] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York, 1994.