An Indexing Scheme for Structured Documents and its Implementation

Tuong Dao
Department of Computer Science
RMIT
GPO Box 2476V
tuong@mds.rmit.edu.au

Ron Sacks-Davis
Faculty of Applied Science
RMIT
GPO Box 2476V
rsd@mds.rmit.edu.au

James A. Thom
Department of Computer Science
RMIT
GPO Box 2476V
jat@cs.rmit.edu.au

Abstract
Documents consist of components such as sections, subsections and paragraphs. These components conform to hierarchical structures which can be explicitly encoded according to document representation standards such as SGML. The SCL (Simple Concordance Lists) query language provides a simple and effective way to express traditional boolean queries as well as queries specifying document structure in a database of structured documents such as SGML documents. This paper describes an indexing scheme which can support this query language and its implementation as a prototype. We describe how SCL queries are evaluated and show how query evaluation can be optimised. The performance results obtained show that structured queries at all levels of the document structure can be supported, and that our optimisation strategy can substantially reduce query processing costs.

Keywords Indexing method, structured document, text database, SGML.

1 Introduction
Documents consist of components such as sections, subsections and paragraphs. These components conform to one or more hierarchical structures which can be explicitly encoded by mark up tags according to document representation standards such as SGML [11]. Typically, two parallel hierarchical structures can be encoded in a book: a hierarchy of logical components such as chapters, sections, subsections, paragraphs and sentences, and a hierarchy of physical components such as pages, columns and lines.

A database system for structured documents should be able to capture the encoded structure and support queries on both contents and structure [1, 5, 15, 16]. Retrieval of components at all levels of the document hierarchy should also be supported as the appropriate unit of retrieval varies with the user needs, it may be a line, a paragraph, a page, a chapter or the entire document.

Traditional information retrieval systems [18] only focus on the contents of documents, they do not take advantage of the structural information present in documents and do not support queries on structure. Recently a number of models of text databases have attempted to integrate both contents and structure in text retrieval. A survey of these models by Baeza-Yates and Navarro [3] provides a description of seven “novel” models representative of the different approaches, and an analysis of these models in terms of expressiveness and efficiency.

Three of the seven models described have linear query time complexity: the Hybrid Model [2], PAT Expressions [17] and GCL (Generalized Concordance Lists) [7, 8]. The Proximal Nodes model [15] has linear query time complexity for most operations; a few operations that are not linear in theory are linear in most practical situations. Considering the remaining three models, query time complexity is $O(n \log n)$ for the List of References model [13], $O(n^2)$ for the Parsed Strings model [10], and non-polynomial for the Tree Matching model [12].

A comparison of the four models with linear or near linear query time complexity in terms of expressive power [14] ranks the Proximal Nodes model first in expressivity, followed by GCL, PAT Expressions and the Hybrid Model, in that or-
der. We consider the first two of these models in more detail. Both Proximal Nodes and GCL place restrictions on their structural model, thus restricting its expressive power, to keep query time complexity linear (GCL), or near linear (Proximal Nodes). Proximal Nodes removes all overlapping of structural elements in the query answers returned, whereas GCL removes all nesting of structural elements in the query answers returned [3, 14]. However we have shown that by extending the GCL model, it is possible to handle both nesting and overlapping in query answers with linear query time complexity [9]. In this paper we describe the implementation and the performance of SCL (Simple Concordance Lists), the extended GCL model.

In Section 2 we provide a brief description of the SCL model and introduce the corresponding text model and the SCL query language. In Section 3 we describe an indexing scheme which can support the SCL query language and its prototype implementation; we describe how SCL queries are evaluated and how query evaluation can be optimised, and provide experimental results. We conclude with a summary of the results obtained and a discussion of further issues.

2 The SCL model

Like the GCL model, the SCL model provides both independence from a defined document hierarchy, and independence from a defined markup schema. It manipulates text intervals and uses containment relationships rather than hierarchical relationships to support queries on document structure. As the model allows nested extents in its data type, called the SC-list (Simple Concordance list), it can handle recursive structures such as lists of lists, which are not supported by the GCL model.

We describe the basic features of the SCL text model and the SCL query language below. In our query examples in this section we will use the collection of two short documents in Figure 1 as our database. The first document is an extract from the novel Confidence by Henry James. The second document is a list of selected works by the same author. Structural elements are explicitly marked up with SGML tags. Each element is delimited by a start-tag of the form <markup-tag> and an end-tag of the form </markup-tag>. For example titles are delimited by the start-tag <title> and the end-tag </title>. The numbers above words and tags represent their positions within the text collection.

2.1 The text model

The text database is a string of concatenated symbols drawn from a text alphabet and a stoplist alphabet. Documents are marked up with symbols drawn from a markup alphabet, each symbol in the markup alphabet corresponds to a structural element name. The index function IT maps each symbol in the text alphabet to the set of integer positions in the database string where the symbol appears. The index function IM maps each structural element name to the set of position pairs which represent the structural elements delimited by corresponding markup symbols. In our example database, with the first digit in each number representing the document number, we have:

\[
\begin{align*}
I_T(\text{proof}) &= \{130, 138, 160\} \\
I_M(\text{quotation}) &= \{(102.5, 133.1), (133.6, 140.1), (142.1, 162.1)\}
\end{align*}
\]

The index function IT and IM are invoked when SCL queries are evaluated. Markup symbols are assigned rational numbers so that they do not interfere with proximity search on text symbols. Symbols in the stoplist alphabet are not indexed but they occupy positions in the database string and serve to maintain proximity relationships.

2.2 The query language

Boolean queries at all levels of the document structure (such as queries 1 and 2 below), and queries on document structure (such as query 3) can be expressed in SCL.

1. Find documents containing "proof" AND "confidence".
2. Find paragraphs containing "confidence" OR "independence".
3. Find items contained in lists which contain "Novels".

The SCL model uses a single data type, the SC-list. Query operators in the SCL query language operate on SC-lists and return SC-lists.

2.2.1 SCL data type

The result of an SCL query is a set of ranges or extents in the database that satisfy the query. Each extent is of the form \((p, q)\) where \(p\) is the start position and \(q\) is the end position.

An SC-list is an ordered set of extents of the form \((p, q)\), where \(p \leq q\). Extents in the SC-list are ordered in ascending order with respect to \(p\) and extents with the same value \(p\) are ordered in descending order with respect to \(q\). Both nesting and overlapping are allowed in an SC-list.

The SCL query \(Q\) below returns the SC-list \(R\), which is the set of extents containing both "proof" and "confidence": The extents in \(R\) overlap at positions 133, 138 and 140.
102.5  103 104 105 106 107 108 109 110 111  112 113 114 115 116 117 117.1
<quotation>... He had all the air of it. He certainly had not the air of a </quotation>
117.2  110 110 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133
<line> man who was going to rush off and give you the last proof of his confidence.

133.1  133.2 133.3
</quotation></par></line>

133.4 133.5
<line><par>
133.6  134 135 136 137 138 139 140 140.1 141 142
<quotation>It was not a proof of confidence. </quotation> said Angela.
142.1  143 144 145 146 147 147.1
<quotation>It had nothing to do </quotation>
147.2  148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
<line> with me. It was as between himself and you; it was a proof of independence ...

162.1  162.2 162.3 162.4
</quotation></par></line></doc>

Document 2

200.1
<doc>
200.2 200.3 201 202 202.1 202.2
<line><title>Second Document </title></line>
202.3 202.4 202.5 203 204 205 206 207 207.1 207.2
<line><list><label>Selected works by Henry James</label></line>
207.3 207.4 208 208.1 208.2 209 210 210.1 210.2
<line><item>Novels</item> <list><item>The Ambassadors</item></line>
210.3  210.4 211 211.1 211.2 211.3 211.4
<line> Confidence</line>
211.5  211.6 212 212.1 212.2 213 214 215 216 216.1 216.2
<item>Criticism</item> <list><item>The Art of Fiction</item></line>
216.3  216.4 217 218 218.1 218.2 218.3 218.4 218.5 218.6
<item>Partial Portraits</item> <list></list></line></doc>

Figure 1: Documents in the text collection
Query Q below returns the SC-list R, which is the set of extents corresponding to the recursive element “list”. The last two extents in R represent the two sublists which are nested inside the enclosing list represented by the first extent.

**Q**: `{all of ("proof", "confidence")}

**R** = `{{(130,133),(133,138),(138,140),(140,160)}}`

2.2.2 SCL operators

SCL operators are listed in Table 1. The simplest SCL query is a quoted string or a tagged-interval. In the query syntax, each *query-expression* is a quoted string, a tagged interval, or another SCL query; a *query-expression-list* consists of one or more *query expressions* separated by commas. The combination operators all of and one of are used in conjunction with the containment operators to express the equivalent of boolean queries (queries 1, 2 and 4 below). The reduction operators derive their names from the data type of the original GCL model, the GC-list [7]. The reduction operator `to-gcl_max` removes the embedded extents in an SC-list, retaining only the “maximal” extents (query 7); and the reduction operator `to-gcl_min` removes the enclosing extents in an SC list, retaining only the “minimal” extents (query 8).

SCL queries corresponding to boolean queries at document level (queries 1-3), and at arbitrary levels of the document structure (query 4), and SCL queries on document structure (queries 5-8) are shown below.

1. **Find documents containing “proof” AND “confidence”**.
   ```
   [doc]
   containing
   (all of ("proof", "confidence"))
   ```

2. **Find documents containing “confidence” OR “independence”**.
   ```
   [doc]
   containing
   (one of ("confidence", "independence"))
   ```

3. **Find documents which do NOT contain “proof of independence”**.
   ```
   [doc]
   not containing
   "proof of independence"
   ```

4. **Find paragraphs containing “proof” AND “confidence”**.

5. **Find lists**.
   ```
   [list]
   R = `{{(202.4,218.4),(208.1,211.2),(212.1,218.2)}}`
   ```

6. **Find items contained in lists which contain “Novels”**.
   ```
   [item]
   contained in
   ([list] containing "Novels")
   R = `{{(207.4,211.3),(208.2,210.1),(210.4,211.1),(211.6,218.3),(212.2,216.1),(216.4,218.1)}}`
   ```

7. **Find items contained in lists which contain “Novels”, remove embedded items**.
   ```
   to-gcl_max (item contained in ([list] containing "Novels")
   R = `{{(207.4,211.3),(211.6,218.3)}}`
   ```

8. **Find items contained in lists which contain “Novels”, remove enclosing items**.
   ```
   to-gcl_min (item contained in ([list] containing "Novels")
   R = `{{(208.2,210.1),(210.4,211.1),(212.2,216.1),(216.4,218.1)}}`
   ```

3 The prototype

First, we describe how SCL queries are evaluated and how we propose to optimise query evaluation. We then describe the indexing scheme that we implemented to support SCL queries and the proposed optimisation of query evaluation. We provide our experimental results and evaluate the performance of the prototype.

3.1 Evaluation of SCL queries

Complete algorithms for the evaluation of SCL operations can be found in two papers [7, 9]. To give a general idea of how SCL queries are evaluated, we show how conjunctive boolean queries are evaluated in the SCL model and how it differs from the evaluation of similar queries in a standard text database. Conjunctive boolean queries consist of a series of search terms connected by the boolean operator AND. As they allow the selection of a small number of relevant documents from a large text collection, conjunctive queries are in general more useful than disjunctive queries which often result in very large result sets.
A standard text database uses a set of inverted lists [4] stored on disk to support boolean queries. Each inverted list corresponds to a term in the database, and contains a list of the numbers of the documents containing that term. To process a conjunctive boolean query, the inverted lists corresponding to the terms in the query are read from disk and merged, the resulting list is the intersection of the sets of document numbers in the inverted lists. For example, if the inverted lists for the term "proof" and "confidence" are

\[ I_{\text{proof}} = \{1, 5, 6\} \]
\[ I_{\text{confidence}} = \{1, 2, 5, 8\} \]

then the documents which satisfy the query proof AND confidence are documents 1 and 5. If the user wants to search for paragraphs and not whole documents, the text collection has to be broken into "documents" of paragraph size, and the set of inverted lists for these new "documents" has to be built. The structural relationship between the resulting "documents" is lost. If the documents in the text collection have various hierarchical structures with many structural levels in each hierarchy, it would be costly to build a set of inverted lists for each structural element in order to support searching at all structural levels, as the total size of the inverted lists can be very large. Also to be able to retrieve and display the actual text to the user, there is the additional cost of mapping tables to map the document numbers for the various document types to physical addresses.

The SCL model also uses a set of inverted lists to support boolean queries, each inverted list corresponds to a term or a markup tag in the database. However the inverted list contains the positions of text intervals occupied by the term/markup tag in the database string, and not document numbers; in fact, document boundary is not necessarily recognised. For example, we could have the following inverted lists for the terms "proof" and "confidence", and the tag <quotation>, with term positions represented as text intervals of zero length.

\[ I_{\text{proof}} = \{(130,130), (138,138), \]
\[ \quad (160,160), (522,522), \]
\[ \quad (651,651)\} \]
\[ I_{\text{confidence}} = \{(133,133), (140,140), \]
\[ \quad (211,211), (518,518), \]
\[ \quad (820,820)\} \]
\[ IM_{<\text{quotation}>} = \{(102.5,133.1), (133.6,140.1), \]
\[ \quad (142.1,162.1)\} \]

To find quotations containing "proof" AND "confidence", we use the SCL query below.

\[ \text{[quotation] containing (all of ("proof", "confidence")\]} \]

First, we evaluate (all of ("proof", "confidence")) by merging the inverted lists \( I_{\text{proof}} \) and \( I_{\text{confidence}} \) to produce the set of minimal extents containing both "proof" and "confidence", \( S_1 \).

\[ S_1 = \{(130,133), (133,138), (138,140), \]
\[ \quad (140,160), (160,211), (518,522), \]
\[ \quad (651,820)\} \]
We then merge the set $S_1$ with the inverted list $I_{M<\text{quot}}$ to produce $R$, the set of quotation extents which contain an extent in $S_1$.

$$R = \{(102.5,133.1),(133.6,140.1)\}$$

To find paragraphs which contain both “proof” AND “confidence”, we merge the set $S_1$ with the inverted list for paragraphs $I_{M<\text{par}}$ instead of $I_{M<\text{quot}}$.

The merging process to evaluate the SCL operations above (described in detail elsewhere [7, 9]) is not unlike the merging process to find the intersection of sets of document numbers in a standard text database. The cost of processing a conjunctive boolean query of $n$ terms at any structural level is the cost of merging $n + 1$ inverted lists, $n$ inverted lists for the $n$ terms, and an inverted list for the structural element. A structural element with a frequency comparable to an unstopped term can be considered as an extra query term to be processed. However for frequently occurring structural elements such as paragraphs or sentences, which have the frequency of a stop word, the extra cost of processing the inverted list for the structural element is potentially very high.

3.2 Optimisation of SCL query evaluation

In order to support boolean queries at all structural levels with a reasonable response time, we need to have a strategy to reduce the high processing cost of frequently occurring structural elements such as paragraphs. We consider below the evaluation of conjunctive boolean queries at paragraph level.

If we recognise document boundary and restrict the merging process of inverted lists for query terms to combining only extents contained within the same document, the extents generated cannot span document boundaries. The set of documents that can contain solutions is non increasing, only extents within this set of candidate documents are processed, all other extents can be ignored. By identifying the query term with the smallest document frequency at the start of query evaluation and reading it from disk, we can establish the smallest set of candidate documents at the beginning and reduce it as more terms are processed, the evaluation process stops if there are no documents left in the set. As the structural element paragraph has high document frequency, it is processed last, after all the query terms have been merged. If the query terms are selective and the remaining set of candidate documents is fairly small, only small parts of the inverted list $I_{M<\text{par}}$ contain extents belonging to the set of candidate documents. It is possible to read from disk and process only a fraction of the inverted list $I_{M<\text{par}}$. If we have a “map” of the inverted list which allows us to locate the parts which contain the paragraphs in the set of candidate documents. The inverted lists in our indexing scheme are stored in contiguous blocks which can be located and fetched individually, making it possible to reduce data transfer and CPU costs by fetching and processing only the required portions of each inverted list.

3.3 The indexing schema

The main components of our indexing scheme are the text lexicon and the text inverted lists, the markup lexicon and the markup inverted lists, and the document map and the word map. To support our optimisation strategy, the text inverted lists and the markup inverted lists are organised in contiguous blocks of variable size. In our implementation, the two lexicons and the document map are kept in memory during query time, the inverted lists and the word map are stored on disk and are read in as required.

The text lexicon (Figure 2) contains at least one entry for each word in the database, excluding stop words, it also functions as an “index map”. Each entry in the text lexicon corresponds to a block in the text inverted lists and includes the following fields: the word itself, the first word position in the corresponding inverted list block, the number of documents in the block, the number of word positions in the block, and the pointer to the corresponding inverted list block. The difference of the pointer values in two consecutive entries gives the size of the block.

Each text inverted list corresponds to a word in the database, it is stored in one or more contiguous blocks and holds a sequence of increasing word positions in the format shown below.

$$\{<\text{docno}>,<\text{numpos}>,\{<\text{pos}>\}...\}...$$

The word positions are preceded by the corresponding document number and the number of positions for that document in that block. The word positions of a document can span consecutive blocks. Blocks have varying sizes which do not exceed a defined maximum block size. If an inverted list is smaller than the defined maximum then it occupies a block of its exact size. Several blocks may be required to store a large inverted list, with a corresponding entry in the text lexicon for each block. Thus we have a map of each inverted list in the text lexicon, the first word position values in two consecutive text lexicon entries tell us the position range and hence the document range of the positions in each block of the inverted list. Our index map is similar to the index map described by Clarke et al. [6], but applied to each inverted list rather than to the entire inverted file.

The markup lexicon and the markup inverted lists are organised in the same way as the text lexicon and the text inverted lists. The markup
lexicon contains an entry for each element name in the database, excluding element names which will not be used in queries. Each entry in the markup lexicon contains the following fields: the element name, the start position of the first position pair in the corresponding inverted list block, the number of documents in the block, the number of position pairs in the block, and the pointer to the corresponding inverted list block.

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

{< docno >< numpospairs > {< pospair >}...}...

Each position pair represents the start position and the end position 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.

The document map and the word map (Figure 3) are used to map any valid position pair to its corresponding physical addresses in the text database. They are used to retrieve the actual text fragments which correspond to the extents of the SC-list returned by the query evaluation process. Each entry in the document map has two fields, the first field represents the offset for the document in the word map, and the second field represents the offset for the document in the text database. Each entry in the word map holds a single value, the document offset for the \((nK + 1)\)th word in each document, \(n\) is a positive integer, \(n > 0\), and \(K\) is a constant. In our prototype, we used the value \(K = 300\) to store the document offset every 300 words. To locate any word or tag position in the actual text, we scan forward for at most 300 words from the offset fetched from the word map.

3.4 Experimental results

We measure the effectiveness of our indexing scheme for conjunctive boolean queries at three structural levels, document, division and paragraph.

Our test database is the collection of English classic texts available for public use at the Oxford Text Archive, the collection includes works by authors such as Henry James and Mark Twain. Although markup tags encoded in this collection conform to a common SGML Document Type Definition, the tagging is not uniform as it has been carried out by various projects for various purposes of text studies. Texts may be encoded as a hierarchy of recursive divisions \(<\text{div}\>\), or as a hierarchy of divisions qualified by a level number \(<\text{div}1\>, <\text{div}2\>, <\text{div}3\>\)... Our indexing scheme is most appropriate for this text collection which has large documents with many structural hierarchies. The characteristics of the database and the uncompressed index, using a block size of 8 Kbytes and a word map constant \(K = 300\), are summarised in Table 2 and 3. All the tags encoded as well as the text, unstopped and unstemmed,
Figure 3: Document map and word map

Table 2: Test database

| Total Text | 38.44 Mbytes |
| Number of Documents | 66 |
| Average Document Size | 582.48 Kbytes |
| Total Unique Words | 71,354 |
| Total Word Occurrences | 6,785,911 |
| Total Unique Markup Tags | 72 |
| Total Divisions | 2,264 (1,626 + 638) |
| Maximal Divisions | 1,626 |
| Total Paragraphs | 112,216 |

Table 3: Index storage requirements

<table>
<thead>
<tr>
<th>Kbytes</th>
<th>% Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Markup Lexicon</td>
<td>6.76</td>
</tr>
<tr>
<td>Markup Inverted Lists</td>
<td>1,797</td>
</tr>
<tr>
<td>Text Lexicon</td>
<td>1,998</td>
</tr>
<tr>
<td>Text Inverted Lists</td>
<td>24,212</td>
</tr>
<tr>
<td>Document Map</td>
<td>0.54</td>
</tr>
<tr>
<td>Word Map</td>
<td>87.76</td>
</tr>
<tr>
<td>Total</td>
<td>28,082</td>
</tr>
</tbody>
</table>

Table 4: Processing time for a 6-term boolean query

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Seeks</th>
<th>Data Read</th>
<th>Time in secs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kbytes</td>
<td>Blocks</td>
<td>Mbytes</td>
</tr>
<tr>
<td>2</td>
<td>16.0</td>
<td>152.4</td>
<td>306.07</td>
</tr>
<tr>
<td>4</td>
<td>14.6</td>
<td>83.3</td>
<td>326.20</td>
</tr>
<tr>
<td>8</td>
<td>12.3</td>
<td>47.9</td>
<td>361.04</td>
</tr>
<tr>
<td>16</td>
<td>10.7</td>
<td>29.4</td>
<td>415.59</td>
</tr>
<tr>
<td>32</td>
<td>8.6</td>
<td>20.0</td>
<td>492.71</td>
</tr>
<tr>
<td>No Blocking</td>
<td>5.0</td>
<td>965.92</td>
<td>0.50</td>
</tr>
</tbody>
</table>

selected paragraph are used to form a query. By selecting query terms from a paragraph in the actual text, we ensure that at least one query answer is returned, and that all the query terms are processed. The results are shown in Figure 4, the values obtained are the average of 5 runs of the set of sample queries, buffers are cleared before each run. The CPU time is the time taken to evaluate the query and return the set of answers, it does not include the time taken to retrieve and display the actual text. Elapsed time is in general 0.16 seconds greater than CPU time. Table 4 shows the results for queries at paragraph level in more detail with, for various block sizes, the average number of seeks issued, the amount of inverted list data read, the CPU time and the elapsed time.

were indexed. We used a Sun Sparc 20 under light load for our experiments, the build time for our database of 38 Mbytes was around 6 minutes.

To measure the time taken to process conjunctive boolean queries, we used a set of 10 sample queries, each consisting of 6 terms. The sample queries are constructed from 10 paragraphs selected at random, the first 6 unstopped terms in each
Figure 4: CPU time to process a 6-term boolean query

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Cost</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kbytes</td>
<td>% Lexicons</td>
<td>% Inv Lists</td>
</tr>
<tr>
<td>2</td>
<td>10.31</td>
<td>0.26</td>
</tr>
<tr>
<td>4</td>
<td>4.65</td>
<td>0.12</td>
</tr>
<tr>
<td>8</td>
<td>2.09</td>
<td>0.05</td>
</tr>
<tr>
<td>16</td>
<td>0.92</td>
<td>0.02</td>
</tr>
<tr>
<td>32</td>
<td>0.40</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 5: Costs and savings of inverted list blocks

Using the text lexicon and the markup lexicon as index maps, we reduce query evaluation costs by fetching only the inverted list blocks which contain extents belonging to the current set of candidate documents, thus realizing a saving of the data transfer and processing cost of unrequired inverted list blocks. At paragraph level we obtained CPU cost reductions of 42% to 60%, using block sizes ranging from 2 Kbytes to 32 Kbytes. At document level and division level there is no significant cost reduction, as each block in these inverted lists contains document or division extents of many documents, it is not likely that unrequired blocks can be isolated.

The reduction of query evaluation costs is achieved at the expense of additional index storage, additional processing cost to identify required/unrequired blocks, and an increase in the number of seeks within each inverted list. Using smaller inverted list blocks increases the size of the lexicons as more lexicon entries are required, but results in greater reduction of evaluation costs than large blocks. As large blocks contain extents belonging to many documents, many more extents which do not belong to the set of candidate documents have to be fetched and processed along with the candidate extents in a required block. Table 5 shows that a smaller block size produces a greater reduction in CPU cost but increases the size of the lexicons, the additional storage required for the inverted list is comparatively small. The cost of identifying the required blocks in inverted lists increases with the number of candidate documents and the number of blocks in each inverted list, this additional cost is not negligible in a large collection of small documents. As portions of the inverted list which are not required are skipped, additional seeks are necessary to position at the next blocks required, however this extra cost is small if the seek distance is small, and a large seek distance would be offset by the saving in processing cost and data transfer for a substantial portion of the inverted list.

Our experimental results show that the blocking scheme implemented can substantially reduce query evaluation processing at the cost of a comparatively small amount of additional storage. As this technique relies on the possibility to isolate and process only the inverted list blocks containing the extents of a small set of candidate documents, it is more effective for conjunctive boolean queries which consist of many terms, and queries on subcollections of documents. However, even for single term queries, using a block size of 8 Kbytes, we obtained a reduction of 23% in CPU cost at paragraph level.
4 Conclusions
The SCL query language supports a wide range of queries on structured documents, including boolean queries at any level of the document structure. Recursive structures, which were not allowed in the earlier GCL model, are freely supported by the SCL model.

We have described an implementation of an indexing scheme which supports the SCL query language and considered its performance for conjunctive boolean queries at different levels of the document structure. For queries on frequently occurring structural elements such as paragraphs, performance can deteriorate compared with queries on less frequently occurring structural elements. To overcome this problem, we proposed and implemented a blocking scheme for inverted lists. This technique, which can be applied to conjunctive boolean queries as well as other types of SCL queries, can substantially reduce evaluation costs for queries on frequently occurring elements; it is particularly effective for queries on subcollections of documents.

Our prototype implementation did not use compression, however the index size can be reduced by compressing the blocks in the inverted lists; in this case pointers in the index maps will point to compressed blocks instead of uncompressed blocks, no other changes to the indexing scheme are required.

Acknowledgements
This work was supported by the Key Centre for Knowledge-Based Systems.

References