BUS: An Effective Indexing and Retrieval Scheme in Structured Documents

Dongwook Shin, Hyuncheol Jang, and Honglan Jin

Department of Computer Engineering
Chungnam National University
220 Kung-Dong, Yusong-Gu, Taejon
305-764 Republic of Korea
Tel: +82-42-821-6657
FAX: +82-42-822-4997
E-mail: {shin, hcjang, hljin}@comeng.chungnam.ac.kr

ABSTRACT

In recent digital library systems or World Wide Web environment, many documents are beginning to be provided in the structured format, tagged in mark up languages like SGML or XML. Hence, indexing and query evaluation of structured documents have been drawing attention since they enable to access and retrieve a certain part of documents easily. However, conventional information retrieval techniques do not scale up well in structured documents.

This paper suggests an efficient indexing and query evaluation scheme for structured documents (named BUS) that minimizes the indexing overhead and guarantees fast query processing at any level in the document structure. The basic idea is that indexing is performed at the lowest level of the given structure and query evaluation computes the similarity at higher level by accumulating the term frequencies at the lowest level in the bottom up way. The accumulators summing up the similarity play the role of accumulating all the term frequencies of the related part at a certain level.

This paper also addresses the implementation of BUS and proves that BUS works correctly. In addition, along with several experiments, it shows that BUS facilitates efficient indexing in terms of space and time and guarantees the reasonable retrieval time in response to user queries.

KEYWORDS: structured documents, SGML, XML, information retrieval, indexing, accumulator

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Digital Libraries 98 Pittsburgh PA USA
Copyright ACM 1998 0-89791-965--3/98/ 6...$5.00

INTRODUCTION

Structured documents are the documents that embed the document structure into the texts. Recently, many documents tend to be produced as structured ones using markup languages like SGML (Standard Generalized Markup Language) [4] or XML (eXtensible Markup Language) [6] since they make it possible to handle the texts in piece by piece in browsing or retrieval. Many digital library systems built up recently assume that the documents have been originally supplied with tagged in SGML. Furthermore, World Wide Web is likely to step toward XML from HTML (HyperText Markup Language) soon in creating Web pages. SGML and XML provide full-fledged features in making documents structured as they are, whereas HTML has only limited functions in structuring.

This tendency calls for the emergency of a new information retrieval system that enables to retrieve and access arbitrary parts of documents (hereafter we use the terminology "element" and "part of document" interchangeably) easily. It raises a difficult problem that the system should be able to figure out relevant elements to users queries issued at any level of the structure, which have not been tackled seriously in the conventional information retrieval system. And most of the structuring techniques proposed so far did not handle the problem efficiently [1, 7, 8, 9, 10].

This paper proposes an indexing and query evaluation scheme (named *BUS* (*Bottom Up Scheme*)) for structured documents that minimizes the indexing overhead and guarantees fast query response time. The basic idea is that indexing is performed at the leaf elements of the given structure and query evaluation computes the similarity at higher level by accumulating the weights at the lowest level in the bottom up way. It underlies the result of R. Wilkinson [15] that *"the retrieval of whole documents can be carried out effectively using just their parts"* in part and the idea of UID (Unique element IDentifier) [5] that enables to compute ancestor element of a given element fast.

The accumulators that accumulate the term frequencies of the corresponding elements play the major role in query evaluation. In the inverted list, a posting is assumed to have the pair of the frequency of a term and the GID (General element Identifier), where the frequency is the number of occurrences in the element and the GID comprises the document number and the UID of an element at the lowest level of the structure with some auxiliary information. In query evaluation, the UIDs in the postings are converted into the corresponding UIDs of the ancestor elements at higher level and the frequencies are added into the accumulators that correspond to those of the ancestors. It results in accumulating all the term frequencies appearing in the ancestor element into the corresponding accumulator. Now, as we reproduce the frequency of a term appearing in an element at an arbitrary level, we can compute the weight of the term in the element in various ways.

This paper also addresses how to implement BUS as effective as possible and proves that BUS works correctly. In addition, according to several experiments, it shows that BUS facilitates efficient indexing in terms of space and time and quick retrieval in response to user queries.

The paper is organized as follows. Section 2 surveys the works concerning about modeling, indexing and retrieval with respect to structured documents. Section 3 proposes the notion of GID (General element IDentifier) and indexing making use of GID. Section 4 presents how query evaluation is done utilizing the indexing information at the leaf nodes. Section 5 addresses the implementation details. Section 6 proves the correctness of BUS and analyzes the performance in terms of indexing overhead and retrieval time. Section 7 presents concluding remarks with some further researches.

RELATED WORKS

For years, there has been growing interest in handling structured documents well in terms of indexing and retrieval. I. Macleod [8, 9] proposed a conceptual model for structured documents named Maestro. It supports structured documents and a query language that enables to retrieve any node in the structure based on the context as well as content. In addition, he suggested a way of processing queries, which could not accommodate the weighed search.

R. Winkinson [15] showed that the retrieval of the whole documents could be carried out effectively using their subparts, but not vice versa. He first measured the similarity of the subparts against the query and yielded the similarity of the whole document by manipulating the results of subparts appropriately. M. Volz et al. [14] suggested an object-oriented coupling with OODBMS that models structured documents into classes. They could compute the similarity of the entities whose indices have been built in advance, but left to the users how to figure out the similarity of the entities whose indices have not been made.

B. Lowe et al. [7] presented the subtree model that is suitable in representing hierarchically structured documents and processing queries. However, they did not suggest a good indexing and retrieval model that avoids duplicated indices and relevant match at any level of the hierarchy. T. Arnold-Moore et al. [1] proposed the ELF model and SGQL query language. ELF model gives a uniform interpretation of the transformation made by the SGQL query language that allows a wide range of structured queries as well as content queries. But, they did not address how to implement the transformation efficiently.

Zprise [12] is one of the public domain software developed by NIST that is able to handle SGML documents and supports Z39.50 protocol. However, it offers a limited range of field search in that it only retrieves elements in a fixed set of fields such as 'title' and 'author', where they should not be hiearchical with one another. Panorama made by Softquad offers a simple search function that is able to highlight the words matched in the query. Even if it gives high quality browsing of SGML documents, it does not have indexing and retrieving features. So far, several commercial systems have been released to support SGML documents. But, most of them (for example the systems developed by OpenText and FernTree) assume to have duplicated indices to each level where the elements are retrieved.

G. Navarro and R. Baeza-Yates [10, 11] suggested a model named Proximal nodes that is expressive and can be efficiently implemented. It first computes the content part of the query and next treats the structural part. They presented an efficient way to compute the structured query as a mapping of a set of nodes to another. However, they did not handle the weighting scheme in the model.

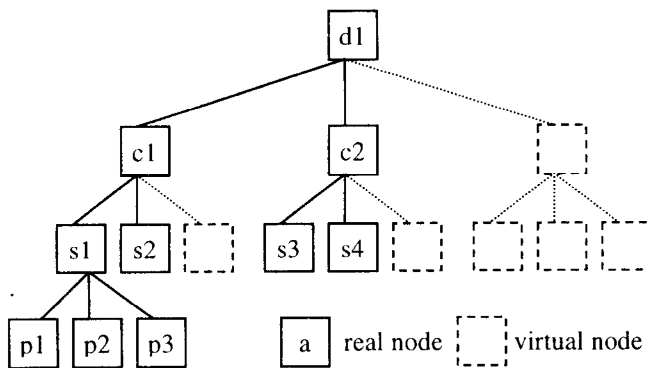
Lee et al. [5] proposed an indexing structure that is able to reduce the storage overhead taken to indexing at all levels of document structure. They first represented a document as a *k-ary* complete tree where k is the largest number of child elements of an element in the structure. The result of the mapping is called '*document tree*'. Secondly they assigned each element a UID (Unique element IDentifier) according to the order of the level-order tree traversal. In this tree, with the knowledge of a child's UID one can compute the parent UID directly by the following expression:

$$parent(i) = \left\lfloor \frac{(i-2)}{k} + 1 \right\rfloor$$

For instance, Figure 1 shows a document tree and the result of assigning UIDs to elements. In this figure, the nodes in dotted line represent the virtual nodes which do not exist in the document tree, but are necessary for making the structure *k-ary* complete tree.

The main idea behind this is that if every sub-element of an element has the same keyword, say, *hypertext*, the keyword need only to be indexed at the parent level, which can take off all the indices at the sub-element level. It was demonstrated to reduce the indexing overhead in the exact

query match, while it does not work in the partial match any more where the source of similarity comes from the weights of terms.



(a) 3-ary document tree

element	UID	element	UID
d1	1	s3	8
c1	2	s4	9
c2	3	p1	14
s1	5	p2	15
s2	6	p3	16

(b) Result of assigning UIDs

Figure 1: Document tree and the result of assigning UIDs

INDEXING STRUCTURED DOCUMENTS IN BUS

Indexing on structured documents can be performed in various ways. A simple method is to index at all levels and to produce appropriate indexing information at the level the query wants. But as it repeats indexing at each level, it usually takes excessive amount of space and time in indexing. Instead of this simple indexing, we carry out the indexing only once at the *text level* and reproduce all the indexing information at higher level in query evaluation. Here the *text level* means the level of an element where the text is included.

General element identifier

The UID proposed by Lee et al. [5] assumes a simple document structure that allows only one type of element at a level. That is, as it assigns only a unique number to each element according to the level-order tree traversal, it does not provide a way of discriminating elements of different types at the same level. For instance, assume that a document structure has the elements, 'title', 'abstract' and 'chapter' at the second level as in Figure 2. Now suppose that a user wants to retrieve documents having 'query evaluation' in the title element. With only UID, there is no way of discriminating the title elements from chapter elements, which might result in retrieving all the documents having 'query evaluation' in 'title' or 'abstract' or 'chapter' element.

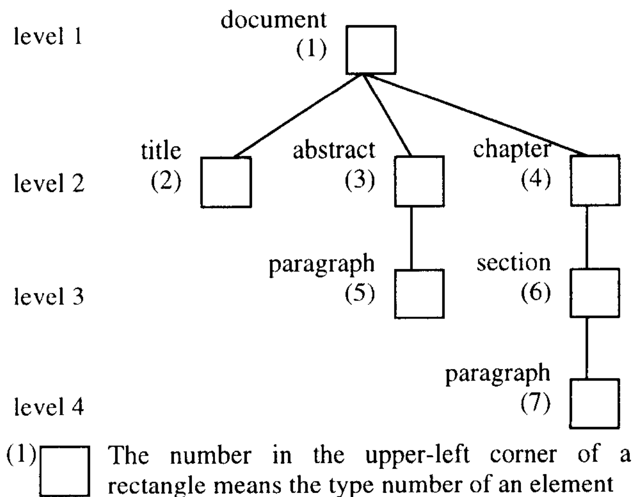


Figure 2: A document structure having different types of elements at a level

As a way of supplementing the limitation of UID, we propose the notion of GID (General element IDentifier) that is composed of (1) Document number, (2) the UID of the element in the document tree, (3) the level of the element in the document tree, and (4) the element type number in the structure. The first constituent in GID informs which document the element belongs to and the second tells the location of the element in the document tree. The third and the fourth constituent facilitate the reproduction of term frequencies in the appropriate level that a user wants. That is, with the third constituent, we can compute the difference of the *user level* (the level that the user wants to retrieve elements) and the *text level* (the levels of the elements where the text is included and thus indexing is really performed), and map the elements at the text level to the elements at user level.

The fourth constituent helps to filter out the elements of types different from what the user wants. For instance, if a user wants to retrieve 'chapter' elements having 'accumulator', the 'title' and 'abstract' elements should not be involved in the computation. As we encode the element type in GID, we can let only the elements of the appropriate types take part in the query evaluation.

Indexing with GID

The indexing is carried out in each element at the text level and terms are extracted with the auxiliary information: (1) the frequency of a term appearing in the element and, (2) the GID of the element. For instance, suppose that a document has the structure with the auxiliary information as shown in Figure 3.

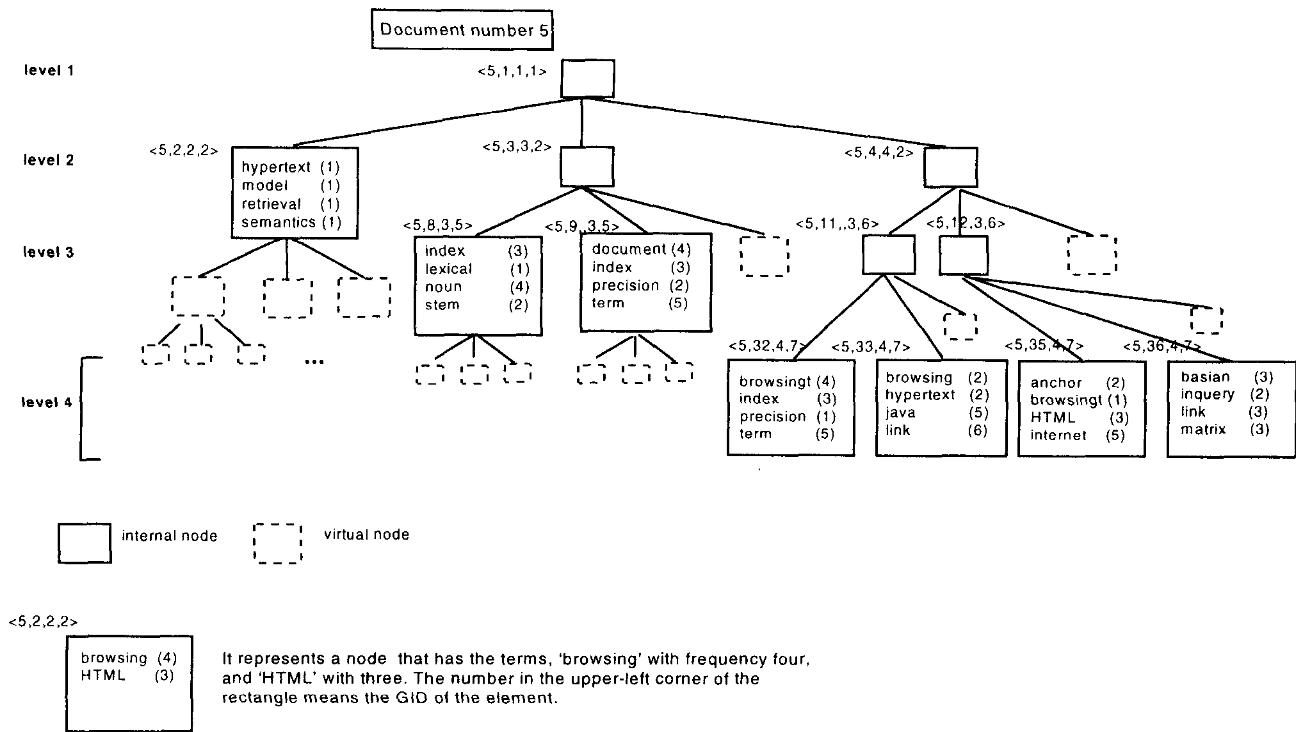


Figure 3: Example document tree with auxiliary information

First, the indexer scans through the document, assigning a GID to each element. Secondly, it extracts terms and calculates their frequencies in each element at the text level. With the setting in Figure 3, the indexer computes the triples: <hypertext, 1, <5,2,2,2>>, <model, 1, <5,2,2,2>>, <retrieval, 1, <5,2,2,2>>, and <semantics,1,<5,2,2,2>> in the first element at the text level, where the first element is the term extracted, the second is the frequency in the element, the third is the GID of the element

As we mentioned before, the indexer carries out the indexing only once scanning through the elements at the text levels. The result of indexing is stored as the inverted index with the B+tree and the posting file.

QUERY EVALUATION OF BUS

With structured documents, it is natural that a user wants to retrieve a part of document at a certain level. For instance, she or he may want to get paragraphs or sections relevant to 'structured document processing'. Query evaluation has to satisfy the queries like this using the information obtained in the indexing step.

Accumulation of frequency

The query evaluation procedure (QEP) of BUS is able to do that by manipulating accumulators and UID gotten from the postings. First, it creates a set of accumulators corresponding to all the elements in the document set. Note that the pair of document number and UID uniquely identifies any element in the document set. Secondly analyzing the user query, it figures out which level and element type the user wants. For instance, if a user issues a query "find out the section containing "browsing" " to the documents having the structure in Figure 2, the QEP recognizes that the user wants to retrieve at level 3 and the element type that should be involved in the query evaluation is 'section' or 'paragrph' in 'section'.

Thirdly, QEP accesses the posting files and extracts the postings. If a posting has a GID whose element type number is six or seven, it calculates the difference of the *user level* and *text level*. For instance, with a posting <browsing, 4, <5,32,4,7>>, the QEP understands that the posting should be involved in the query evaluation and calculates the difference of the *user level* and *text level* - one. Fourthly, the QEP maps the UID in the posting to the parent UID at the user level, creating UID 11 from UID 32 using the formula presented in Section 2. Fifthly, the frequency 4 is added into the accumulator <5,11>, where the first constituent represents the document number and the second means the UID. Doing this way, the QEP sums up all of the frequencies of the descendant elements to the accumulator corresponding to the user level element. Figure 4 shows the result of accumulators.

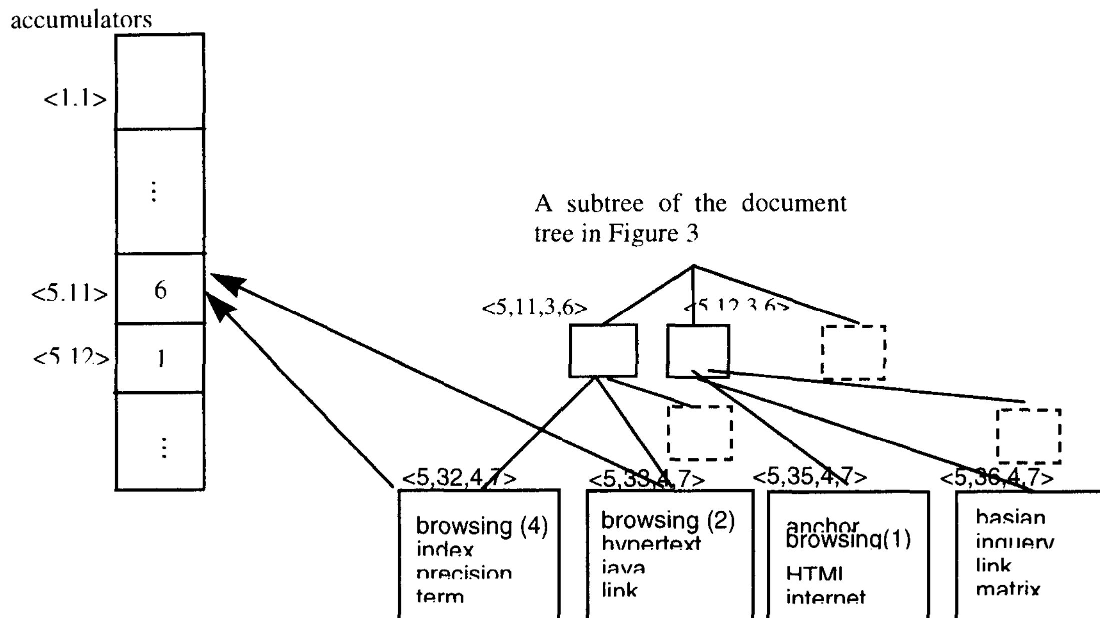


Figure 4: The result of the accumulators

Weight computation with term frequency and document frequency

Once the frequency of a term is obtained, its weight can be computed. Normally, the weight of a term is calculated by using the term frequency in the document, the IDF (Inverse Document Frequency) of the term, and a normalization function. Among these, the IDF value is computed as a function of the document frequency, which is acquired as the number of non-zero accumulators in BUS. Table 1 shows several variations of these components.

Table 1: Components of term weighting scheme

Term Frequency Component		
b	1.0	binary weight equal to 1 for terms in a document (term frequency is ignored)
n	tf	raw term frequency (number of times a term occurs in a document)
a	$0.5+0.5\frac{tf}{\max tf}$	augmented normalized term frequency (use maximum normalization where each tf is divided by maximum tf, and further normalize the resulting value to lie between 0.5 and 1.0)
l	$\ln tf + 1.0$	logarithmic term frequency which reduces the importance of raw term frequency in those collections with widely varying document length
Document Frequency Component		
n	1.0	no change in weight; use original term frequency component (b, n, a, or l)
t	$\ln \frac{N}{n}$	multiply original term frequency component by an inverse document frequency factor (N is the total number of documents in the collection, and n is the number of documents in which a terms appears)
Normalization Component		
n	1.0	no change; use factors derived from term frequency and collection frequency only (no normalization)
c	$\frac{1}{\sqrt{\sum_{vector} w_i^2}}$	use cosine normalization where each term weight w_i is divided by a factor of Euclidian vector length

As for the term frequency, there has been suggested several ways to take it to weight computation. Among these, we can calculate most of formulas directly from frequency information except the augmented normalized term frequency. The augmented normalized term frequency calls for the knowledge of the maximum frequency of the terms appearing in the target element, which can not be kept track of in BUS. It is because the frequency of a term at the target element is accumulated at query evaluation time when the term is issued by a user. Hence, there is no way of knowing which term occurs most frequently in the target element without accumulating frequencies for all the terms appearing in the collection.

With regard to IDF, we can reproduce the raw document frequency as the number of non-zero accumulators in each term retrieval step. The IDF is obtained as a simple logarithmic function of the raw document frequency as shown in Table 1.

As for the normalization factor, we can not apply the cosine normalization since it requires the knowledge of all weights of terms in the target element. It is owing to the same reason that we can not compute the augmented normalized term frequency. However, we have no reason to stick to cosine normalization, since it has been reported that byte length normalization yields significant improvement in retrieval effectiveness over cosine normalization [13]. Byte length is easily computed as the summation of all the byte lengths of the *text level* elements that belong to the target element.

IMPLEMENTATION

The key issues in implementing BUS are : how to represent the posting structure and how to manage accumulators efficiently. In traditional IR systems, a posting is composed of document id, term frequency and so on. In BUS, a posting has the same structure as the one used in traditional IR except that GID is used instead of document id. With respect to memory management, BUS requires as many accumulators as the number of elements in the document set. However if we use the hashing technique we can reduce the number of accumulators significantly.

As described above, a posting structure is drawn in Figure 5. Here, the UID consumes eight bytes since in handling complicated DTD, UID occasionally grows too big.

DID	UID	Level	E_type_num	Tf

DID : Document ID
UID : Unique element ID
Level : Element level
E_type_num : Element type number
Tf : Term frequency

Figure 5: Posting structure in BUS

In managing accumulators, BUS basically calls for as many accumulators as the number of elements in the document set. To reduce the memory consumed at run time, we use hashing techniques. The memory structure managing accumulators are described in Figure 6.

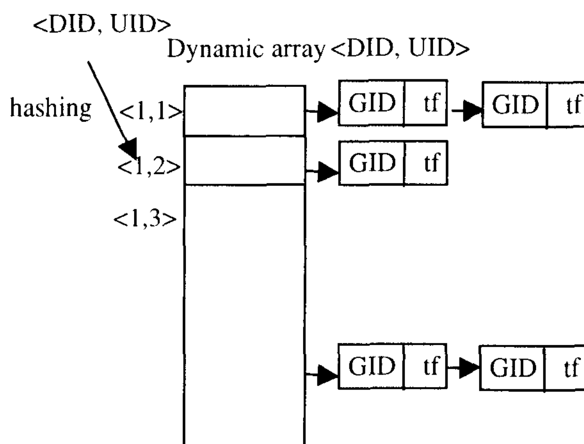


Figure 6: Memory structure for accumulators

In Figure 6, when a posting is read from the posting file, the <DID, UID> pair is mapped into an element of the dynamic array using a hash function. If an accumulator collides with another, it is chained at the end of the linked list.

BUS has been implemented in C/C++ on Solaris operating system. The system works on client-server model, where the client is programmed in JAVA applets. Hence, anyone can freely connect to the system in a Web browser and test how it works. At present, BUS is able to handle SGML documents, but can support documents in XML easily, if the SGML parser is replaced by the XML one.

ANALYSIS

Correctness of BUS

In this section, we prove that BUS works correctly and demonstrate that it is feasible. Note that the main objective of BUS is to index the structured documents only at *text level* and reproduce the data necessary for computing the weight at *use level* dynamically.

First, Theorem 1 proves that BUS reproduces the raw data used in computing weights - term frequency and document frequency exactly.

Theorem 1. BUS reproduces exactly the term frequency and the document frequency of the element that the query wants.

Proof) Suppose that a query wants an element type '*ele*' at level k and the text has been indexed at level l. QEP in section 4 extracts only postings from the posting file whose level is l, and merges the frequencies into the accumulator that corresponds to the element at level k.

Now suppose that there is an element which is a descendant of '*ele*', but whose frequency is not added to the corresponding accumulator of the ancestor '*ele*'. Then there are two possibilities: first is that the element is not indexed, nor kept into the posting file. Second is that the frequency is not added to its ancestor accumulator. But the two cases are not possible because all the elements must have been indexed in BUS and the frequencies of elements at level l should be added to the corresponding accumulator at level k by the UID calculation. Therefore, the accumulators must have all the term frequencies of the corresponding element.

Secondly, the document frequencies are calculated as the number of accumulators whose value is non-zero. It is apparent because if there is an element that has at least one occurrence of a term, the term should appear in a sub-element of the element at level l. Then the frequency must be added to the corresponding accumulator, which makes the value of the accumulator greater than zero. (Q.E.D.)

Now we prove that we can reproduce the term frequency and document frequency of an element at *user level*. Hence, we can yield the same retrieval effectiveness in six weighting schemes (3 term frequency variations x 2 document frequency variations) among sixteen weighting schemes described in Table 1. We are not able to reproduce other ten weighting schemes exactly. But, in six out of ten that use cosine normalization, we are able to yield at least the same retrieval effectiveness as the original ones if we employ byte length normalization [13] instead of cosine normalization.

The remaining four schemes are those which use augmented normalized term frequency as the term frequency component. In these four, we can not reproduce the same effectiveness, nor guarantee the similar result as the original ones.

Experimental result of BUS implementation

Here, we analyze how BUS indexes and retrieves efficiently as compared with the traditional IR technique. As an experimental data, we take three sub-collections in TREC corpus [3] – PATENT, AP, and WSJ. In particular, we choose Patent sub-collection in measuring query evaluation since it allows nested structure (mean the structure where an element is contained in another element) and has deeper structures than others do. Appendix 1 shows the DTD structure of Patent collection graphically. In this structure, the element '*Doc*' represents the whole document, while '*Text*' and '*Par*' indicate certain parts of the document. Note that the traditional IR system is only capable of retrieving texts at '*Doc*' level, whereas BUS allows the retrieval at any level specified in the DTD.

Table 2 shows the indexing overhead of BUS in three sub-collections. After indexing the original collection, we apply one of the compression method named δ- coding [16] to each field in the posting structure separately.

Table 2: The indexing overhead of BUS

collection	data size (M byte)	index size (M byte)			time (hour)
		before compression	after compression	index overhead after compression (%)	
PATENT	256	284.59	119.66	46.74	2.5
AP	254	369.81	62.36	24.55	2.4
WSJ	267	346.98	60.87	22.80	2.6

As shown in Table 2, BUS consumes around 23 to 47 % space overhead after compression and takes around 2.5 hours in indexing about 250 M bytes. This overhead is never significant because with 20 to 50 % more space, we can handle every element residing in each document. However, if we apply the traditional IR method in handling structured documents and index the documents that have deep nested structure, the index size is likely to grow too big reaching several hundred percents (even several thousand percent) of the size of the real data. Moreover, it may take huge amount of time in indexing because the indexer repeats indexing at each level in the whole hierarchy.

Note that the compression ratio in PATENT sub-collection is lower that the other two. It is because the UID values created in PATENT are far greater than those made in AP or WSJ. In fact, PATENT has a deep hierarchical structure, whereas AP and WSJ do not. As we move toward to lower levels in the document structure, the UIDs of elements increase fast. The effect of the compression is diminished when the target values are high.

In the aspect of retrieval time, BUS does not add much overhead because it only calls for calculating the UID of the target element in retrieving each posting, which could be computed by a multiple of two addition, one division and one truncation operation (It is because we have to repeat the computation of finding parent UID as many times as the difference of the *user level* and *text level*.)

Table 3 shows the response time of BUS and traditional IR respectively, when a user query is given to the PATENT collection. Table 3 summarizes the response times of 50 queries issued at three levels : *Doc*, *Text*, and *Par* level as shown in Appendix 1. The queries are made manually from the concept fields in TREC queries - 51 to 100. In query evaluation at *Doc* or *Text* level, the UID (located at *Par* level) in each posting is converted to the UID at user level (*Doc* or *Text* level), while at *Par* level, the UID need not be converted. As expected, the retrieval can be performed in a

reasonable speed. The average response time of the fifty queries is 1 second in *Doc* level and 0.54 second in *PAR* level. Note that the response time in *DOC* level is longer than that in *TEXT* or *PAR* level. It is because the text is indexed in *PAR* level and the level difference from *DOC* to the text level is greater than the other two. As opposed to BUS, the traditional IR system simply extracts postings and goes through similarity calculation.

Table 3: The comparison of BUS and the traditional IR system with respect to the response time

Topic Num	Query made from concept fields \ Result of retrieval	Traditional IR method		BUS (element type number, level)					
				DOC (1,1)		TEXT (63,2)		PAR (1850919,5)	
		num	time (sec)	num	time (sec)	num	time (sec)	num	time (sec)
057	MCI Communications Corp.	14	0.07	14	0.24	14	0.23	2	0.24
063	batch, interactive, process, user interface	4145	0.26	4145	2.73	4142	1.97	11670	0.98
065	storage, database, data, query	2300	0.16	2300	2.36	2279	1.80	4456	1.13
067	students, agitators, dissidents	338	0.02	338	0.53	338	0.51	269	0.47
068	fine-diameter fibers, glass, ceramic, mineral-wool, asbestos, cellulose	446	0.09	446	0.54	445	0.50	408	0.48
075	increased efficiency, smaller payroll, work force reduction	5135	0.35	5135	2.47	5131	1.84	10646	1.22
077	poaching, illegal hunting, fishing, trapping, equipment	448	0.10	448	0.75	415	0.74	337	0.69
082	genetically engineered product, plant, animal, drug, microorganism, vaccine, agricultural product	3281	0.33	3281	4.18	3189	3.25	9973	2.22
083	Greenhouse effect, global warming, carbon dioxide buildup	4653	0.24	4653	2.71	4648	2.09	9268	1.40
096	diagnosis, scanning, testing	2033	0.11	2033	1.40	2006	1.16	2703	0.78
Average of the ten queries			0.17		1.79		1.41		0.96
Average of 50 queries (TREC query 051-100)			0.09		1.01		0.88		0.54

CONCLUSION AND FURTHER STUDIES

Structured documents have been gaining growing attention since most of documents in digital libraries are beginning to be made with tagged in SGML. Furthermore, it is likely that many Web documents are going to be written in XML instead of HTML. However, the conventional information retrieval systems and the structural methods proposed so far do not handle the weighting scheme well in the document structure.

This paper suggests BUS (Bottom Up Scheme) - an efficient indexing and query evaluation method for structured documents. and shows that it indexes and retrieves structured documents effectively. The basic idea is that indexing is performed at the lowest level of the given structure and query evaluation reproduces the term weights at higher level by accumulating the term frequencies at the lowest level in the bottom up way. The accumulators summing up the similarity play the role of accumulating all the weights of the related part at a certain level.

This paper also shows how to implement BUS efficiently and demonstrates that it indexes and retrieves structured documents efficiently. An experimental result with TREC collection shows that BUS does not add much overhead in indexing phase in terms of space and time, and retrieves any elements in a reasonable speed.

We implemented BUS on Solaris operating system and demonstrated that it worked quite well. At present, BUS can handle SGML documents. But it is able to accommodate XML documents easily if the SGML parser is replaced by the XML one. Therefore, we believe that BUS is one of the promising methods in constructing an information retrieval system in the future digital libraries or Word Wide Web.

Several works remain to be done. First, the UID values hardly change once it is assigned to an element. However, if a document is updated and exceed the largest number of chid

nodes – k, which is decided in the old one, the UIDs should be computed again. Secondly, we will apply the BUS to the various kinds of structured query languages [1, 2, 9] which provide a mixed form of content and structure.

ACKNOWLEDGEMENTS

We are grateful to Hyoungsik Woo, Youngil Kim and Hyojin Nam for their invaluable work in developing various parts of the software.

REFERENCES

1. Arnold-Moore, Fuller, T. M. Lowe, B Thom, J and Wilkinson, R. The ELF data model and SGQL query language for structured document databases, Proc. Sixth Australian Database Confgerence, 1995.

2. Dao, T. Sacls-Davis, R. Thom, J.A. Indexing Structured Text for Queries on Containment Relationships.

3. Harman, D. Overview of the Second Text Retrieval Conference, Proc. The Second Text Retrieval Conference (TREC-2) (1994) pp 1-20.

4. Herwijnen, E. Practical SGML: Second Edition, Kluwer Academic Publishers, 1994.

5. Lee, Y.K. Yoo, S.J. Yoon, K. Berra, P.B. Index Structures for Structured Documents," Proc. Digital Library '96 (1996) pp. 91-99.

6. Light, R. Presenting XML, Sams Net, 1997.

7. Lowe, B. Zobel, J. Sacks-Davis R. A Formal Model for Databases of Structured Text, Proc. DASFAA'95 (1995).

8. Macleod, I.A. Storage and Retrieval of Structured Documents, Information Processing and Management 26, 2 (1990) pp. 197-208.

9. Macleod, I.A. A Query Language for Retrieving Information from Hierarchical Text Structure, The Computer Journal 34, 3 (1991) pp. 254-264.

10. Navarro, G. A Language for Queries on Structure and Contents of Textual Databases, Master Thesis, Unuversity of Chile, 1995.

11. Navarro, G and Baeza-Yates, R. Proximal Nodes: A Model to Query Document Databases by Contents and Structure, ACM transaction on SIGMOD, 1996.

12. NIST, Guide to Z39.50/PRISE 1.0, NIST document, 1995.

13. Singhal, A. Salton, G. and Buckley, C. Length Normalization in Degraded Text Collection, Technical report in Cornell University, 1995.

14. Volz, M. Aberer, K. Bohm, K. A Flexible Approach to Combine IR Semantics and Database Technology and its Application to Structured Document Handling, GMD Technical Report No. 891, 1995.

15. Wilkinson, R. Effective Retrieval of Structured Documents, in Proc. the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval (1994) pp. 311-317.

16. Witten, I.A. Moffat, A., and Bell, T.C. Managing Gigabytes, Van Nostrand Reinhold, 1994.

Appendix 1: The DTD structure in PATENT sub-collection