29th Hawaii International Conference on System Sciences, vol. 2, pp. 155-164, Maui, Hawaii, January 1996

Querying Structured Hyperdocuments *

Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon

School of Computer & Information Science
Syracuse University
Syracuse, NY 13244-4100

P. Bruce Berra

Dept. of Electrical & Computer Eng.
Syracuse University
Syracuse, NY 13244-4100

Abstract

In this paper, we present a document model which integrates the logical structure and hypertext link structure of hyperdocuments in order to manage structured documents with hypertext links. Based on this model we define a new structure query language which expresses the structure query using path expressions.

To process a structure query in a document management system which represents structure information as database relations, costly join operations are used to find a relationship between elements in a document hierarchy. In order to overcome this problem, schemes based on the parse tree [6] and element locator [2] have been used. In this paper, we propose a new structure query processing scheme that uses unique element identifiers (UID's) to evaluate structure queries. Our scheme has advantage over previous schemes since it can obtain the UID's of the ancestors and descendents directly from the UID of a node without disk access. We present relational database schemas for our scheme as well as others and compare the query processing costs.

In order to support direct access to a document element, keyword indices to it should be provided. We propose three kinds of inverted index structures for efficient structure query processing.

1 Introduction

Since the SGML [16] was standardized, one of the most important issues in research has been the storage and retrieval of structured documents using database systems. There has been much research effort in the field of structured document management and handling including [2] [6] [8] [10] [21]. To store documents into databases, many issues should be addressed such as the granularity of data to be stored, structure query language, indexing scheme, and query processing. In this paper, we provide solutions for some issues including document modeling, query language, database schema, element indexing, and query processing.

SGML, HTML [5], and HyTime [15] documents contain the hierarchical structure as well as hypertext links. These structures are complementary and by combining them we can build a more powerful document management system [10]. Our query language is based on the document model which integrates the hierarchical logical tree structure of documents and hypertext links. Our query language supports two types of queries: content query and structure query. The content query is based on the content of documents. For example, the query that finds documents which contain a specific keyword is a content query. The structure query can be a simple structure query that can be resolved by using DTD's (Document Type Definition) only or a more complex structure query which is combined with the content query. For example, the query which finds documents that have specific elements such as chapter and section is a simple structure query. In the complex structure query, users can find sections whose first paragraph contains a specific keyword.

Our structure query language uses the concept of document structure addressing. We have considered three kinds of addressing methods: list addressing, tree addressing, and hypertext addressing. We use path expressions in the query to express these addressing methods. The path expression has been used in [10]. Their path expression supports only forward directional navigation. We have extended the path expression to represent backward paths from lower levels to higher levels in the document tree and backward hypertext links. Thus, users can navigate in either

*This work has been supported in part by the Electronics and Telecommunications Research Institute of Korea and the New York State Center for Advanced Technology in Computer Applications & Software Engineering.

direction freely and select any interested portion of a document or a group of documents. We also have extended the expression to be able to represent the number of hops by a method similar to the regular expression in automata theory. By using our path expression, users can easily control the number of hops of branches or links in the document tree or hypertext graph, and express powerful structure queries.

To process structure queries, we need document structure information. Previously, database relations [8], parse trees [6], and element locators [2] have been used. However, they require considerable space and disk access time. In this paper, we propose a novel scheme which uses specially designed unique element identifiers (UID's) for structure queries. By letting the UID carry information about the document structure, the UID's of the ancestor or descendent nodes can be obtained directly from the UID. Thus, we can perform structure queries very fast without disk access.

In order to perform structure queries efficiently, an index structure which supports fast element access should be provided. However, there has been little research on the index structures for structured documents [21]. In this paper, we present new inverted index structures which facilitate structure query processing.

2 Related Work

Several hypertext reference models have been proposed including the Dexter model [13] in order to provide a systematic basis for the system and to develop interchange standards. The Dexter model treats the within-component layer as being outside its scope. Thus, the SGML or ODA model can be used in the within-component layer in conjunction with the Dexter model [3]. Our model differs from the Dexter model. In order to fully support structure queries, we need to represent the document structure of the within-component layer in the model. Thus, the goal of our model is to represent the document logical structure as well as the hypertext links.

Supporting search and query in a hypermedia network has been an important issue that should be addressed in the hypermedia systems [12]. Many query languages have been proposed. For example, Beeri and Kornatzky [4] have proposed a logical query language, Amann and Scholl [1] have presented a graph data model and query language, and Schutt and Streitz [24] have defined a hypertext query language called HTQL. However, they have not considered the logical document structure. Christophides et al. [10] have combined the hierarchical logical structure and hypertext links and proposed a structure query language based on path expressions. Structure queries can be easily generated by using the path expression. We have extended their path expression concept.

There have been many debates as to whether hypermedia storage systems should be based on the database paradigm or file system paradigm [19]. According to Lange [17], a hypertext system should be an application of database systems in order not to reinvent the last 20 years of database technology. Some hypermedia systems have already been developed on top of the database management systems. Intermedia [28], HyperBase [24], and HB2 [23] are examples. Document systems using database systems have also been studied in [2] [6] [8] [10] [20] [21] [25]. In this paper, we present a relational database schema because most commercial database systems adopt the relational data model and it has enough functionality to implement our system. However, our scheme can be applied to other models with some changes.

In order to support document structure queries, we have to maintain information about document structures. The relational schema proposed in [8] can be used for this purpose. This scheme stores elements into different relations and represents the parent-child relationship between elements using unique element surrogates. However, this approach requires heavy join operations during structure query processing because the structure information is scattered into several relations.

The parse tree scheme proposed in [6] avoids relational join operations during structure query processing by grouping all structure information together in a parse tree. Even though it does not require join operations, it requires much space to store parse trees. Moreover, the parse trees related to the required documents must be accessed each time to process the structure query, which requires considerable disk access time.

In [2], another scheme has been proposed which does not require join operations either. Each element is assigned a unique identifier and it is associated with a LOC (the location of the element within the document). It uses the DTD as the database schema and processes simple structure queries by using only the DTD. To process complex structure queries, it has to obtain the LOC of an element to find the location of the element and the UID's of the ancestors and descendents. The problem is that it requires another disk access because the UID's of the ancestors and descendents cannot be obtained directly from the UID and LOC of an element. In Section 5, we present a new scheme which overcomes this problem.

Much research has been carried out to design efficient index structures for database and information retrieval systems [14] [22]. Index structures for hypertext systems have been proposed in [7] [9]. These are useful for selecting interested nodes from a hypertext network. However, they have not considered the document logical structure within a node. The objective of our index is to support fast keyword access to document elements. Sacks-Davis et al. [21] have proposed some inverted index structures for structure query processing. However, their schemes require considerable storage overhead. In this paper, we propose index structures for structured documents that reduce the storage overhead considerably.

3 Structured Hyperdocument Model

The document model, represented as a graph, forms a conceptual document structure on which a query language is based. The document graph is defined as follows:

- The *document graph* consists of a set of nodes N and a set of edges E between pairs of nodes. Here a node can be any element of a document such as a section or paragraph.
- E has two types of arcs: One represents the hierarchical logical structure of a document, and the other represents hypertext links between nodes within a document or between documents. The first type of arc forms a document tree, and the second type forms a directed hypertext graph.

An example of the document graph is illustrated in Figure 1.

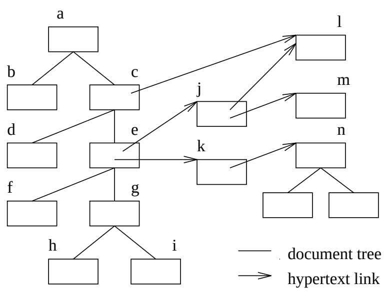

Figure 1: Example Document Graph

4 The Structure Query Language

Our query language [18] has a similar format to the SQL. In the query, users specify required documents using a document expression. The document expression is defined as follows.

- A document expression is of the form $d.P$, where d is a document variable and P is a path expression or a group of path expressions connected by a dot(.).
- A path expression can be a branch expression, link expression, or list expression.

4.1 Document Tree Addressing

Tree addressing is used to navigate from the current position to some other relevant positions of the document. It uses two kinds of branch expressions: forward branch expression and backward branch expression.

The forward branch expression is defined as follows.

- branch[*] : nodes reachable by 0 or more forward hops in the document tree.
- branch[+] : nodes reachable by 1 or more forward hops.
- branch[0] : current node.
- branch[>] : all leaf nodes of the current node.
- branch[i] : nodes reachable by i hops forward with i a positive integer.
- branch[i - j] : nodes reachable by the forward hops in the range given by i and j with i, j positive integers and $i < j$.

The backward branch expression is defined as follows.

- branch[/] : nodes reachable by 0 or more backward hops in the document tree.
- branch[-] : nodes reachable by 1 or more backward hops.
- branch[<] : root node.
- branch[-i] : node reachable by i hops backward with i a positive integer.
- branch[-i - -j] : nodes reachable by the backward hops in the range given by i and j with i, j positive integers and $i < j$.

The following shows some examples of the branch expression for Figure 1.

```
e.branch[*]={e,f,g,h,i}  e.branch[/]={a,c,e}
e.branch[2]={h,i}        e.branch[-2]={a}
e.branch[1-2]={f,g,h,i}  e.branch[<]={a}
```

It is obvious how some of these expressions can help find document elements whose leaf nodes have a particular keyword. The more complex structure queries also are useful. Example queries using the branch expression are given below.

- Find sections that have a subsection containing the keyword "hypertext".

```
find s
from a:article, s:a.section
where exists (s.branch[1] contains "hypertext")
```

- Find siblings and children of this_element.

```
find e.branch[1],
   difference(e.branch[-1].branch[1],e)
from e:this_element
```

4.2 Hypertext Addressing

The link expression is designed to give complete addressing capability to the hypertext system. The link expression is defined similarly as the branch expression.

The forward link expression is defined as follows.

- link[*] : nodes reachable by 0 or more forward hops in the hypertext graph.
- link[+] : nodes reachable by 1 or more forward hops.
- link[0] : current node
- link[>] : all leaf nodes of the current node.
- link[i] : nodes reachable by i hops forward with i a positive integer.
- link[i - j] : nodes reachable by the forward hops in the range given by i and j with i, j positive integers and $i < j$.

The backward link expression is defined as follows.

- link[/] : nodes reachable by 0 or more backward hops in the hypertext graph.
- link[-] : nodes reachable by 1 or more backward hops.
- link[<] : all root nodes of the current node.
- link[-i] : nodes reachable by i hops backward with i a positive integer.
- link[-i - -j] : nodes reachable by the backward hops in the range given by i and j with i, j positive integers and $i < j$.

The following shows some examples of the link expressions for the document graph of Figure 1.

```
e.link[+]={j,k,l,m,n}    l.link[-]={c,e,j}
e.link[2]={l,m,n}        l.link[-2]={e}
e.link[1-2]={j,k,l,m,n}
```

The following shows query examples using the link expression.

- Find the images referenced by the section of an article containing the keyword "hypermedia".

```
find i
from a:article, s:a.section, i:image
where (s contains "hypermedia")
      and (i in s.link[1])
```

- Find the elements that refer to the third subsection of the last section of my_article.

```
find e
from a:my_article, e:a.element,
     ss:a.section[>].subsection[3]
where ss in e.link[1]
```

4.3 List Addressing

List addressing is used to locate a subelement of an element from the document tree. The list expression is defined as follows:

- L[i] is a list expression, where L is an element name, branch expression, or link expression, and i is an integer or character '>'. It returns the i-th element, or the last element in the case of '>', from the list L.

Examples of the list expression are given below.

```
section[3]: third section
section[2].paragraph[>]: last paragraph
             of the second section
branch[1][2]: second child of a node in the tree
```

The operator precedence of the list expression is left associative. For example, for the expression branch[1][2], we first evaluate branch[1], then we select the second element from the result. Query examples using the list expression are given below.

- Find the title and the first author of articles having a section containing the words "database" and "hypermedia".

```
find a.title, a.author[1]
from a:article
where a.section contains
      ("database" and "hypermedia")
```

- Find the last paragraph of the second section of this_article.

```
find a.section[2].paragraph[>]
from a:this_article
```

4.4 Combination of Addressing Methods

The branch expression, link expression, and list expression can be combined to make more powerful path expressions. The following examples show combined expressions for the document graph in Figure 1.

```
a.branch[2][2].link[1]={j,k}  e.link[1][2].link[-1]={e}
l.link[-1][1].branch[1]={d,e} e.branch[-2][1]={b}
k.link[-1].branch[+]={f,g,h,i}
```

The following is an query example.

- Find the title of articles that are referenced by the last child of this_element.

```
find a.title
from a:article, c:this_element
where a in c.branch[1][>].link[1]
```

5 Structure Query Processing

In this section, we propose a novel scheme which can process structure queries very fast.

5.1 Element Identifier Coding Scheme

We interpret a document structure as a k-ary tree [11] and assign each element a UID according to the level-order traversal order. For example, for the document tree of Figure 2, we assign UID's as Table 1 assuming a 3-ary tree. Because we consider the tree complete, there are some virtual nodes which do not exist. Using these UID's we can easily get the UID's of the parent and child nodes of a specific node.

The UID's of the parent and j-th child of a node whose UID is i can be obtained by the following functions.

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

$$child(i,j) = k(i-1) + j + 1 \qquad (2)$$

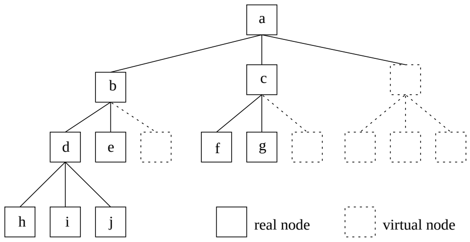

Figure 2: Example Document Tree

Table 1: Unique Element Identifiers for Figure 2

element	UID	element	UID
a	1	f	8
b	2	g	9
c	3	h	14
d	5	i	15
e	6	j	16

5.2 Hypertext Navigation

In order to support hypertext navigation, we use the join index [26] that will be discussed in Section 7. If the hypertext graph is acyclic, the evaluation of link expressions is not difficult even though complicated link expressions such as link[*] require much time. When the hypertext graph is cyclic, we do not generate a UID more than once during the evaluation of link expressions to avoid an infinite loop.

6 Database Schema

We design a relational schema that supports our coding scheme (CDS). In order to compare our scheme with other schemes, we have also designed database schemas for the parse tree scheme [6] and the element locator scheme [2]. Figure 3 is the DTD for a sample article and Figure 4 is a document that conforms to the DTD. We will use these for explanation.

```
<!doctype ARTICLE [
<!element ARTICLE   - - (TITLE, AUTHOR, ABSTRACT, SECTION+))>
<!element TITLE     - O (#PCDATA)>
<!element AUTHOR    - O (NAME)+>
<!element ABSTRACT  - O (#PCDATA)>
<!element SECTION   - O (TITLE, PARAGRAPH+)>
<!element NAME      - O (#PCDATA)>
<!element PARAGRAPH - O (#PCDATA)+>      ]>
```

Figure 3: Sample DTD for Article

```
<ARTICLE>
 <TITLE>
  Structured Hyperdocument Query Language
 <AUTHOR>
  <NAME> Y. K. Lee
  <NAME> P. B. Berra
 <ABSTRACT>
  We describe a query language for structured
  documents.
 <SECTION>
  <TITLE>
   Introduction
  <PARA>
   We define the path expression.
 <SECTION>
  <TITLE>
   Path Expression
  <PARA>
   Branch expression exploits the logical structure
   of documents.
  <PARA>
   Link expression is used for hypertext navigation.
</ARTICLE>
```

Figure 4: Sample SGML Marked Document

6.1 Parse Tree Scheme (PTS)

In this scheme, a parse tree is associated with each document to represent structure information. We also need relations in order to store leaf nodes in the database. For the sample document in Figure 4, we need the relations as shown in Table 2, Table 3, Table 4, Table 5, and Table 6.

Table 2: Relation TITLE

document id	element id	data
1	2	Structured Hyperdocument ...

A sample parse tree of the example is shown in Figure 5 using a binary tree. In order to process simple structure queries, we need another data structure to represent the relationship between elements as shown in Figure 6. For database access, we need an element mapping table as shown in Table 7.

Table 3: Relation NAME

document id	element id	data
1	12	Y. K. Lee
1	13	P. B. Berra

Table 4: Relation ABSTRACT

document id	element id	data
1	4	We describe a query ...

Table 5: Relation SECTION.TITLE

document id	element id	data
1	22	Introduction
1	27	Path Expression

Table 6: Relation PARAGRAPH

document id	element id	data
1	23	We define the path ...
1	28	Branch expression exploits ....
1	29	Link expression is used for ....

Table 7: Element Mapping Table

element number	element name	relation name
1	article	-
2	title	TITLE
3	author	-
4	abstract	ABSTRACT
5	name	NAME
6	section	-
7	section.title	SECTION.TITLE
8	paragraph	PARAGRAPH

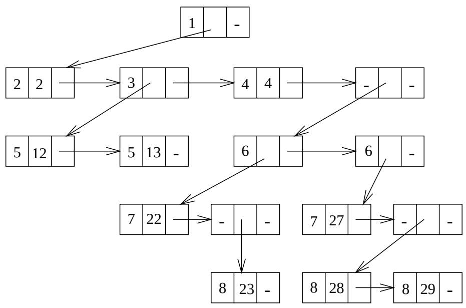

Figure 5: Sample Parse Tree

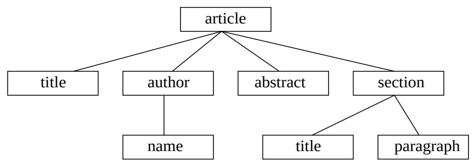

Figure 6: Element Hierarchy Tree

6.2 Element Locator Scheme (LOS)

For this scheme, we can use the relations defined in the PTS. Instead of the parse tree, we need a table which associates an element with an LOC (location within a document). Thus, the LOC table should contain the pair (*element identifier*, *LOC*) for all elements in a document. This scheme also needs the element hierarchy tree as shown in Figure 6.

6.3 Coding Scheme (CDS)

In the CDS, we use the same relations as the PTS. Instead of the parse tree, a table is maintained for each document to represent the number of children of the internal node. This is used during query processing to know how many children each internal node of a document has. The number_of_children_table for the sample document is presented in Table 8. We also need the element hierarchy tree as shown in Figure 6.

Table 8: Number of Children

element id	number of children
1	5
3	2
5	2
6	3

6.4 Cost Comparison

We have analyzed the space requirements and disk access times of the PTS, LOS, and CDS in [18]. Here we summarize the results of the analysis.

6.4.1 Storage Requirement

The PTS, LOS, and CDS use the same database relations for storing document elements. Besides these, the PTS maintains the element_hierarchy_tree, element_mapping_table, and parse tree. The LOS maintains the element_hierarchy_tree and LOC table. The CDS maintains the element_hierarchy_tree and number_of_children_table. The element_mapping_table and element_hierarchy_tree are maintained for each DTD. Because the database will not contain many DTD's, we can ignore the storage cost for these tables.

The parse tree in the PTS, LOC table in the ELS, and number_of_children_table in the CDS are maintained for each document. Thus, the cost of storage

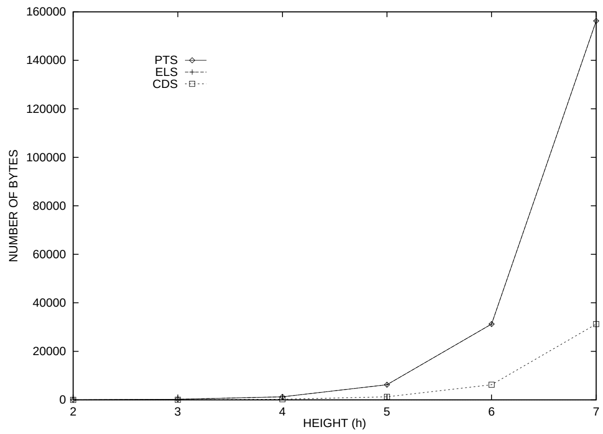

Figure 7: Storage Requirement

depends on these tables. Figure 7 shows the space requirements of the PTS, LOS, and CDS, when the document tree is a full 5-ary tree. The CDS shows the best storage utilization while the PTS and LOS show quite similar results (same curve in the graph). It shows that the difference increases as the height of the tree increases.

6.4.2 Disk Access Time

Forward Branch Expression: In the PTS, we have to access parse trees as many times as the set size, that is, the number of elements, of the document variable in the forward branch expression. For example, in order to evaluate d.branch[1], the number of parse trees to be accessed is the set size of d. In the LOS, the LOC table should be accessed as many times as the set size of d. Similarly, in the CDS, the number_of_children_table should be accessed as many times as the set size of d.

Figure 8 shows the disk access times of the PTS, LOS, and CDS for the forward branch expression, when the degree of the tree is 5 and the set size is 10 (with 28 msec random block access time, 2 msec sequential block access time). The CDS shows the best disk access time while the PTS and LOS show similar results.

Backward Branch Expression: In the PTS, the number of parse trees which should be accessed to evaluate a backward branch expression is the same as the set size of d. In the LOS, the disk access time is the same as that of the forward branch expression. However, in the CDS, we do not need to access the disk because we can get the UID's of the ancestors by the *parent* function.

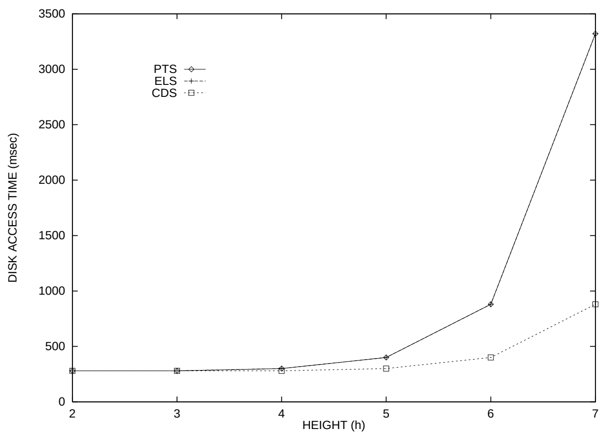

Figure 8: Disk Access Time

Link Expression: In order to evaluate link expressions, join indices [26] are maintained for all the PTS, LOS, and CDS. Thus, the time required to evaluate the link expression is the same for all schemes.

List Expression: In the PTS, parse trees should be accessed as many times as the set size of the document variable. In the LOS, the number of LOC table to be accessed is the same as that of the PTS. However, in the CDS, we do not need to access the disk.

7 Index Structures

Suppose that a document has three leaf nodes and each leaf has keywords as shown in Figure 9. Even though an internal node has no associated data, the data of its subtrees should be considered as its data. Thus, the index for each node is as follows:

```
index(A) = {person,female,man,girl,woman,animal}
index(B) = {person,female,girl,woman,animal}
index(C) = {person,man,animal}
index(D) = {person,female,girl,animal}
index(E) = {person,female,woman}
```

7.1 Inverted List for All Nodes with Replication (ANWR)

The first naive approach is to replicate all keywords in the children to their ancestor nodes. In this scheme, the inverted list for a keyword should include all UID's of the elements which contain it.

```
list(person)={A,B,C,D,E}  list(female)={A,B,D,E}
list(girl)={A,B,D}        list(woman)={A,B,E}
list(man)={A,C}           list(animal)={A,B,C,D}
```

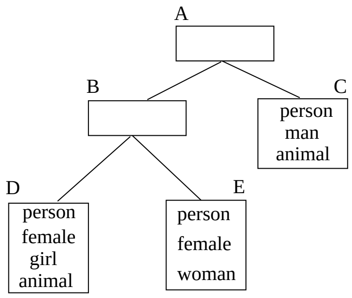

Figure 9: Document Tree with Keywords

Using the inverted list we can access any element at any level in the document tree. However, this scheme causes many duplications in the inverted list.

7.2 Inverted List for Leaf Nodes Only (LNON)

In this scheme, we include element identifiers in the inverted list for leaf nodes only.

```
list(person)={C,D,E}  list(female)={D,E}
list(girl)={D}        list(woman)={E}
list(man)={C}         list(animal)={C,D}
```

Even though we do not maintain the UID's of the internal nodes in the document tree, the UID's of the ancestor nodes can be calculated using the *parent* function.

7.3 Inverted List for All Nodes without Replication (ANOR)

In this scheme, we use the fact that the children nodes of a node can have some keywords in common. By using this fact, we can construct the inverted list for the example as follows:

```
list(person)={A}  list(female)={B}
list(girl)={D}    list(woman)={E}
list(man)={C}     list(animal)={C,D}
```

From this inverted list, we can access any element in the database using the *parent* and *child* function. If the number of documents in the database is quite large, the inverted list must contain a large number of document element identifiers. In this case, this scheme can reduce the storage space of the inverted list and disk access time considerably.

7.4 Index Structure for Hypertext Links

In order to support the fast evaluation of the link expression, we need an index structure. In our system, we use the join index [26] to represent hypertext links in the database. The join index consists of six attributes: source document identifier, source element identifier, anchor address, destination document identifier, destination element identifier, and destination address. One B^+ tree is constructed for the pair of source document identifier and element identifier, and another B^+ tree is built for the pair of destination document identifier and element identifier.

8 System Implementation

We are currently implementing the prototype of our structured hyperdocument management system which consists of DTD processor, document processor, and query processor, as shown in Figure 10.

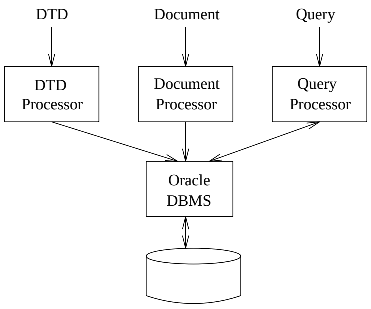

Figure 10: Document Management System

The DTD processor consists of DTD parser, database schema generator, and database mapper generator. The DTD parser parses a DTD and generates a parse tree. For the DTD parser, we have used the Amsterdam SGML Parser [27] and added a parse tree generation module. From the parse tree, the database schema generator generates the database schema for the DTD in SQL. The database mapper generator generates a database mapper which partitions a document into elements and stores them separately into the database.

The document processor consists of document parser, element identifier generator, and index builder. The document parser generates a parse tree for a given document. From the parse tree, the database mapper generates SQL statements which insert document elements into the database. The element identifier generator makes a UID for each element based on our identifier coding scheme, and the index builder extracts keywords and inserts them into the index structure.

The query processor evaluates queries given by the user using the proposed query processing scheme.

9 Conclusions

In this paper, we have defined a new structure query language based on the document model which integrates the logical structure and hypertext structure of documents. Since the query language has been designed to easily navigate the document's logical structure and hypertext structure, users have the advantage of navigating through documents via logical paths and hypertext links and retrieving any interested portions of a document or documents.

In order to process structure queries efficiently, we have proposed a new scheme which uses unique element identifiers. Our query processing scheme is especially efficient for the evaluation of branch and list expressions. For the backward branch expression and link expression, we do not need to access the disk at all. We have compared our scheme with previous schemes and have shown that our scheme has the best performance.

To support fast document element access, we have proposed new inverted index structures which reduce the index storage overhead considerably. According to [18], the ANOR shows the best performance among the three proposed schemes in storage overhead and disk access time.

Our schemes have some limitations. In the design of the link expression, we have not considered link types. We have considered only the untyped links of the SGML documents. However, the link expression can be easily extended to support these. In this paper, we have not supported document sharing and document versioning. In order to support these, we have to change our element identifier scheme. We are considering introducing another level of UID's in order to map the UID into a document element in the database. Addressing these problems is our future work.

Acknowledgments

We would like to thank Michael Bieber and anonymous referees for helpful comments.

References

[1] B. Amann and M. Scholl, "Gram: A Graph Data Model and Query Language," *Proc. of the ACM Conference on Hypertext,* 1992.

[2] T. Arnold-Moore, et al., "The ELF Data Model and SGQL Query Language for Structured

Document Databases," Technical Report 94-13, CITRI, Australia, 1994.

[3] V. Balasubramanian, "State of the Art Review on Hypermedia Issues and Applications," Technical Report, Rutgers Univ., 1993.

[4] C. Beeri and Y. Kornatzky, "A Logical Query Language for Hypertext Systems," *Proc. of the First European Conference on Hypertext,* 1990.

[5] T. Berners-Lee and D. Connolly, "Hypertext Markup Language (HTML) - 2.0," Available via http://www.w3.org/hypertext/WWW/MarkUp/MarkUp.html, June 1995.

[6] G. E. Blake, et al., "Text/Relational Database Management Systems: Harmonizing SQL and SGML," *Proc. of the International Conference on Applications of Databases,* 1994.

[7] P. D. Bruza, "A Novel Aid for Searching in Hypermedia," *Proc. of the First European Conference on Hypertext,* 1990.

[8] D. M. Choy, et al., "Document Management and Handling," *IEEE '87 Office Automation Symposium,* 1987.

[9] C. Clifton and H. Garcia-Molina, "Indexing a Hypertext Database," *Proc. of the 16th VLDB Conference,* 1990.

[10] V. Christophides and A. Rizk, "Querying Structured Documents with Hypertext Links using OODBMS," *Proc. of the European Conference on Hypermedia Technology,* 1994.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms,* The MIT Press, 1990.

[12] F. G. Halasz, "Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems," *CACM,* 37(7), July 1988.

[13] F. G. Halasz and M. Schwartz, "The Dexter Hypertext Reference Model," *CACM,* 37(2), Feb. 1994.

[14] D. Harman et al., "Inverted Files," in *Information Retrieval: Data Structures and Algorithms,* W. B. Frankes and R. Baeza-Yates ed., Prentice Hall, 1992.

[15] ISO/IEC 10744. *Information Technology - Hypermedia/ Time-based Structuring Language (HyTime),* 1992.

[16] ISO 8879. *Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML),* 1986.

[17] D. B. Lange, "Object-Oriented Hypermodeling of Hypertext Supported Information Systems," *Proc. of the 26th Hawaii Int'l Conference on System Sciences,* 1993.

[18] Y. K. Lee, et al, "Structured Document Management and Handling," Technical Report 9506, CASE Center, Syracuse University, June 1995.

[19] J. J. Leggett, et al. ed., "Final Report of the NSF Workshop on Hyperbase Systems," TR TAMU-HRL 93-003, Texas A&M University, July 1993.

[20] I. A. Macleod, "Text Retrieval and the Relational Model," *Journal of American Society for Information Science,* 42(3), 1991.

[21] R. Sacks-Davis, T. Arnold-Moore, and J. Zobel, "Database Systems for Structured Documents," *Proc. of the International Symposium on Advanced Database Technologies and Their Integration (ADTI '94),* 1994.

[22] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval,* McGraw-Hill, 1983.

[23] J. L. Schnase, *HB2: A Hyperbase Management System for Open, Distributed Hypermedia System Architecture,* Ph.D. Thesis, Texas A&M Univ., 1992.

[24] H. A. Schutt and N. A. Streitz, "Hyperbase: A Hypermedia Engine Based on a Relational Database Management System," *Proc. of the First European Conference on Hypertext,* 1990.

[25] M. Stonebraker, "Docuemnt Processing in a Relational Database System," *ACM Trans. on Office information Systems,* 1, 1983.

[26] P. Valduriez, "Join Indices," *ACM Transactions on Database Systems,* 12(2), June 1987.

[27] J. Warmer and S. Van Egmond, "The Implementation of the Amsterdam SGML Parser," *Electronic Publishing,* 2(2), John Wiley & Sons, July 1989.

[28] N. Yankelovich, et al., "Intermedia: The Concept and the Construction of a Seamless Information Environment," *Computer,* Jan. 1988.