STORAGE AND RETRIEVAL OF STRUCTURED HYPERMEDIA/MULTIMEDIA DOCUMENTS USING DATABASE SYSTEMS

by

Yong Kyu Lee

B.E., Dongguk University, 1986
M.S., Korea Advanced Institute of Science and Technology, 1988

ABSTRACT OF DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer and Information Science
in the Graduate School of Syracuse University

May 1996

Approved Professor P. Bruce Berra

Date February 9, 1996

Abstract

In order to build a hypermedia/multimedia document management system using a database system, many issues related to the storage and retrieval of documents should be addressed. In this research, we provide approaches for some important issues including hyperdocument modeling, query language, database schema, index structure, and query processing.

Hyperdocuments contain a hierarchical structure and hypertext structure that can be integrated to provide more powerful query facilities. We present the Structured Hyperdocument Model (SHM) which combines both structures, and based on the SHM we define the Structured Hyperdocument Query Language (SHQL) that expresses the structure query using path expressions. By using the path expression, users can freely navigate the hierarchical paths and hypertext links in the document and select a node or a group of nodes.

An index structure which supports fast element access must be provided in a structured document management system. However, it requires considerable space if we use one of the conventional index structures. Based on the inverted index and signature file, we propose various kinds of indexing schemes which can reduce the storage overhead considerably. We analyze and compare the storage requirements and disk access times of them.

To process a structure query in a document system which represents the document structure as database relations or parse trees, considerable storage and query processing time are required. In this research, we propose a 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

accessing database relations or parse trees. We present relational database schemas for our scheme and others, and analyze the structure query processing costs of them. Based on the comparison, our scheme has the best performance.

Like text documents, video documents have a hierarchical logical structure of video elements. We present a tree-structured video document model, and based on this model we indicate how to apply the SHQL, index structures, and query processing scheme to video documents.

STORAGE AND RETRIEVAL OF STRUCTURED HYPERMEDIA/MULTIMEDIA DOCUMENTS USING DATABASE SYSTEMS

by

Yong Kyu Lee

B.E., Dongguk University, 1986
M.S., Korea Advanced Institute of Science and Technology, 1988

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer and Information Science
in the Graduate School of Syracuse University

May 1996

Approved Professor P. Bruce Berra

Date February 9, 1996

Acknowledgments

I would like to express my sincere gratitude and appreciation to my advisor Prof. P. Bruce Berra for his advice, encouragement, and active support throughout this research. His knowledge and expert guidance have been invaluable at every stage of this research.

I am grateful to Prof. C. Y. Roger Chen, Prof. Chilukuri K. Mohan, Prof. Carlos R. P. Hartmann, and Prof. Alok Choudhary for serving on my dissertation committee and to Prof. Salim Hariri for chairing the committee.

I also acknowledge the New York State Center for Advanced Technology in Computer Applications and Software Engineering (CASE) at Syracuse University, who supported this research in part by grants from the Electronics and Telecommunications Research Institute (ETRI) of Korea.

Special thanks go to my family for their constant love, understanding, and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the Standard Generalized Markup Language (SGML) [39] [50] was standardized in 1986 by the International Organization for Standardization (ISO), one of the most important research issues has been the storage and retrieval of structured documents. In order to support hypermedia documents, the Hypermedia/Time-based Structuring Language (HyTime) [51] was standardized by the ISO in 1992, and the HyperText Markup Language (HTML) [10] has been used in the World-Wide Web [8] [9] on the Internet. There has been much research effort in the field of structured document management and handling including [6] [11] [12] [13] [17] [21] [22] [24] [26] [27] [28] [42] [55] [59] [60] [66] [71] [79]. To store and retrieve hyperdocuments using databases, many issues should be addressed such as query language, indexing scheme, and query processing. In this research, we provide approaches for some important issues including document modeling, structure query language, database schema, document element indexing, and structure query processing.

There have been two approaches to storing documents in databases. One is to store a whole document as a text string, and the other is to partition a document into component elements and store them separately. The first approach has the advantage when users want to retrieve entire documents. However, structure query processing is difficult because the entire document must always be retrieved and analyzed. To ease the problem, a data structure such as a parse tree can be used. However, whenever users want to retrieve a part of a document, it has to be partitioned before delivery to them. Conversely, in the second approach, join operations are required to restore the entire document. However, structure query processing is easier since the structure information is already stored in the database, and hyperdocuments can be naturally supported. This approach exhibits good performance when parts of documents are frequently retrieved. It can easily support element sharing by multiple documents and handling document versioning is also easy. In this research, we will not consider the problem of selecting one approach because each has its own advantage and disadvantage. Instead, we will concentrate on the problem of identifying the required documents or document elements from the database, and our schemes will be applicable to both approaches.

The Structured Hyperdocument Query Language (SHQL) defined in this research is based on the Structured Hyperdocument Model (SHM) which integrates the hierarchical logical structure and hypertext structure of documents. By combining both structures, the system can provide more powerful query facilities. The SHQL supports the structure query as well as the content 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 document type definitions (DTD's) and a complex structure query which is combined with the content query. For example, the query which finds documents which 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.

The SHQL uses the concept of document structure addressing. We have considered three kinds of addressing methods: tree addressing, hypertext addressing, and list addressing. We use path expressions in the query to express these addressings methods. The path expression has been used in [26] [27]. 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 to either 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 of 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 network, and express powerful structure queries.

Much research has been carried out to design efficient index structures for database, hypermedia, and information retrieval systems [11] [19] [30] [34] [35] [47] [67] [68]. In order to perform structure queries efficiently, an index structure which supports fast element access must be provided. For this purpose, an index structure based on the conventional inverted index or signature file can be used. Sacks-Davis et al. [66] have proposed some possible inverted index structures for structure query processing. However, every scheme requires considerable storage overhead to store document element identifiers in the inverted list. In this research, we propose various kinds of new inverted indexing schemes and signature file schemes which reduce the storage overhead considerably. We analyze storage requirements and disk access times of these schemes.

In order to support document structure queries, we have to maintain information about document structures. The relational schema proposed in [24] can be used for this purpose. This scheme stores elements into different relations and represent the parent-child relationship between elements using unique element surrogates. This scheme assigns a unique surrogate to each element and each database tuple corresponding to the element contains the surrogate of its parent. By using the surrogate, the parent-child relationship between elements can be known. 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 [17] can be used to avoid relational join operations during structure query processing by grouping all structure information of a document together in a parse tree. Even though it does not require join operations, it requires much space to store parse trees in the database. Moreover, the parse trees related to required documents must be accessed each time to process the structure query, which requires considerable disk access time.

In [6], 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) which is similar to the tree location address of the HyQ [55]. 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 this research, we propose a novel scheme which overcomes this problem. Our scheme 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. We present relational database schemas for our scheme as well as others and compare query processing costs.

Recently, much attention has been focused on video databases including [63] [70] [76] [77]. Video documents also have a hierarchical logical structure as with text documents [62] [70]. By exploiting this structure using structure queries, users can obtain greater benefits than by using only content queries. However, there has been little attention to the structure query language in video databases. In this research, we propose solutions for some issues in video databases including structure query language, index structure, database schema, and query processing.

The contents of this research are organized as follows according to chapters: In the next chapter, we survey related work in the field of hyperdocument storage and retrieval. We present the Structured Hyperdocument Model (SHM) that incorporates the hierarchical logical structure and hypertext structure of hyperdocuments in Chapter 3. Chapter 4 defines the Structured Hyperdocument Query Language (SHQL) based on the SHM. In Chapter 5, we propose a couple of document element identifier coding schemes which can be used to represent the hierarchical document structure. Various kinds of structured document indexing schemes based on the inverted index and signature file are presented in Chapter 6. Chapter 7 provides relational database schemas for our scheme as well as other schemes. An analysis of the structure query processing costs of various schemes appears in Chapter 8. In Chapter 9, we present our video document management scheme. Chapter 10 describes the prototype of the structured hyperdocument management system. Finally, our conclusions and future research appear in Chapter 11.

Chapter 2

Related Work

2.1 Introduction

Documents including text documents, multimedia documents, and hypermedia documents have a well-formed hierarchical logical structure. For example, an article may have several sections, and a section may consist of several subsections, and so on. Hypertext links of the hypermedia documents also form a hypertext network. By exploiting the hierarchical structure and hypertext network of documents, we can build a document management system with powerful query facilities [27]. In order to manage structured hyperdocuments in a database system, many issues should be addressed including structure markup language, structure query language, structure representation, and index structures. In this chapter, we survey previous research related to the structured hyperdocument management.

2.2 Structure Markup Language

To markup the logical structure of documents, the Standard Generalized Markup Language (SGML) [39] [50], the Office Document Architecture (ODA) [4] [49], and the Hypermedia/Time-based Structuring Language (HyTime) [51] [61] have been standardized by the International Organization for Standardization (ISO). For the hypertext documents of the World-Wide Web [8] [9] on the Internet, the HyperText Markup Language (HTML) [10] has been used. In this section, we give a brief introduction to the SGML and HTML since they are most popular markup languages.

2.2.1 The Standard Generalized Markup Language

The SGML was standardized by the ISO in 1986. The document model provided by the SGML is a hierarchical tree-structured model. A class of documents which conform to a document type, such as article, memo, or book, is defined by a document type definition (DTD). Figure 2.1 shows an example of the DTD defined for simple articles. An article consists of a title, one or more authors, an abstract, and one or more sections. A section consists of one or more paragraphs, and a paragraph is defined as a character string. Other elements are defined similarly.

A logical element of an SGML document is located between a *start-tag* and an *end-tag* as defined in the DTD. An element between these two tags may have nested subelements introduced by other tags. Figure 2.2 is an SGML document marked according to the DTD of Figure 2.1.

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

Figure 2.1: Sample SGML DTD

2.2.2 The HyperText Markup Language

The HTML is a hypertext markup language which is used in the World-Wide Web (W3) initiative [8] [9] on the Internet. The HTML 2.0 standard was proposed by the HTML Working Group of the Internet Engineering Task Force (IETF) in 1995, and the HTML 3.0 is under development. The HTML is used to represent the following as listed in [10].

- hypertext news, mail, online documentation, and collaborative hypermedia
- menus of options
- database query results
- simple structured documents with inlined graphics
- hypertext views of existing bodies of information

The HTML DTD is defined in terms of the SGML and all HTML documents must conform to the same DTD. An HTML document contains markups which represent a hierarchy of document elements and a hypertext link structure. Figure 2.3

```
<ARTICLE>
  <TITLE>
    Indexing Structured Documents
  </TITLE>
  <AUTHOR> Y. K. Lee
  <AUTHOR> P. B. Berra
  <ABSTRACT>
    We describe index structures for structured documents.
  <SECTION>
    <TITLE>
      Introduction
    </TITLE>
    <PARA>
      We present new index structures.
  </SECTION>
  <SECTION>
    <TITLE>
      Inverted Index
    </TITLE>
    <PARA>
      We need an index structure for efficient query processing.
    <PARA>
      We propose new inverted index structures for structured
      documents.
  </SECTION>
</ARTICLE>
```

Figure 2.2: Sample SGML Document

```
<HTML>
  <TITLE>
    Querying Structured Hyperdocuments
  </TITLE>
  <BODY>
    <PRE>
      <b>Yong Kyu Lee</b>
      School of Computer and Information Science
      Syracuse University
      yklee@top.cis.syr.edu
    </PRE>
    <H2>Querying Structured Hyperdocuments</H2>
    <P> We present a document model which integrates the logical
    structure and hypertext network 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.
    <UL>
      <LI> <A HREF="hicss.html">Querying Hyperdocuments </A>
    </UL>
  </BODY>
</HTML>
```

Figure 2.3: Sample HTML Document

shows an example HTML document that contains a hierarchical document structure with a hypertext link.

2.3 Structured Document Models

Many document models have been proposed since the standardization of the SGML. Generally, we can classify these models into two categories: tree-structured document models and hypermedia document models.

2.3.1 Tree-Structured Document Models

Most of the structured document models are based on the tree structure. For example, the document models defined in [22] [24] [42] [59] [60] [71] belong to this category. The document management systems based on these models can support queries on the hierarchical structure, but not on the hypertext link structure. However, the structured documents such as the SGML and HTML documents contain the hypertext link structure as well as the tree structure. In order to fully support such documents, the hypertext links should be able to be represented in the model.

2.3.2 Hypermedia Document Models

Many hypermedia data models have been defined including the KMS model [1], Graph Server model [2], HAM model [23], MINOS model [25], Trellis model [37], Garg's abstraction model [38], Dexter model [46], Lange's formal model [56], HyperBase model [69], Hypergraph model [72], and Intermedia model [78]. These models consider a document as a hypertext network of nodes. In this subsection, we survey the Dexter model as it is the de facto standard hypermedia reference model.

2.3.2.1 The Dexter Hypermedia Reference Model

The Dexter hypermedia reference model has been defined in order to provide a systematic basis for the system and to develop interchange standards [45] [46]. In this model, a hypermedia system is divided into three layers: the run-time layer, the storage layer, and the within-component layer, as presented in Figure 2.4. It also defines two classes of operations for the storage layer and run-time layer respectively.

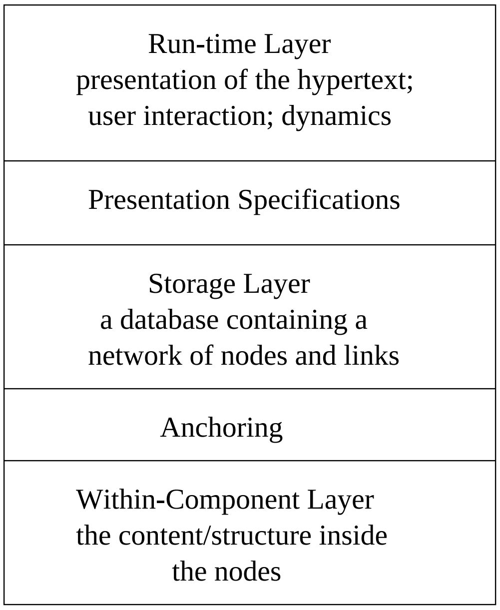

Figure 2.4: Layers of the Dexter Model [46]

Among the three layers, the focus is on the storage layer that models basic structural entities (node, link) and aggregation structures [18], and the run-time layer and the within-component layer are beyond its scope. However, the SGML or ODA model can be used in the within-component layer [7], and thus, this model can be combined with the tree-structured document model.

2.4 Structure Query Language

Sacks-Davis et al. [66] have classified the types of queries which are required to support SGML documents as follows:

1. Word-by-word access
2. Ranked queries
3. Query scope restricted to sub-documents
4. Retrieval of sub-documents
5. Access by structure of documents
6. Access to different types of documents
7. Access by other SGML markup
8. Combined access to SGML and non-SGML data

The first two types of queries are based on the document content. Users can search documents which have some specific keywords using these types of queries. The next three types of queries (from type 3 to type 5) are based on the document structure. Using these queries, users can search document elements rather than the entire document based on the document structure. The type six query is for supporting multiple DTD's in a document management system, and the type seven query is used to locate documents based on document attributes associated by users such as version number and security level. Finally, the last type query is used to access non-SGML documents as well as SGML documents.

Arnold-Moore et al. [5] [6] have presented a data model and query language for the SGML documents. In their model in [6], documents in the database are considered as a list of ELF's (ELements with Features) where an ELF is defined as:

- a complete SGML element
- a list of features associated with that element

And the element features include:

- an absolute element identifier
- a document identifier
- a DTD (document type definition)
- an associated output specification
- a similarity measure
- the location of that element within the document

Based on the ELF data model, they have defined a functional query language, called SGQL (Structured Generalized Query Language), which supports structure queries as well as content queries. Their language provides a set of basic functions to handle ELF's and they can be combined to make more powerful functions. The language is suitable and powerful for simple queries. However, the SGQL is not user-friendly for complicated structure queries because it requires many functions to be combined in a method similar to the low-level functional programming language. Moreover, users should know the procedure of query evaluation since the query language is procedural.

Kimber [55] has presented a query language, called HyQ, for the HyTime [51] documents. He has proposed various kinds of tree addressing methods as follows:

- Tree location address: addresses a node of a tree by concatenating children numbers from the root to the node. For example, the address '1 2 3 4' represents the fourth child of the third child of the second child of the root.
- List location address: selects one or more elements from a nested node list.
- Path location address: addresses one or more paths from a tree by treating it as a matrix in which a column corresponds to a path. Using the column number, any path can be identified.
- Relative location address: locates other nodes relative to a node in a tree.

The HyQ is a functional query language which is similar to the SGQL. The HyQ provides basic functions for simple queries, and they are combined to generate a complex function. It is also a procedural language like the SGQL.

Macleod [59] [60] has defined a query language based on a tree-structured document model. His language has used the concept of path in a document hierarchy. Some example paths are as follows:

```
Paragraph[1] of SubSection[2] of Section[3]
SubSectionHeading of SubSection[4]
ParagraphHeading of Paragraph[5] of SubSection[6]
```

As shown in the above example, a path is a sequence of element names in the document tree. In order to make a structure query, users must know the element names and their hierarchy defined in the DTD. Thus, it is not user-friendly and is difficult to support multiple DTD's in the document management system.

Christophides et al. [26] [27] have extended this path concept and presented a query language, called Extended O_2SQL, which uses path expressions for structure queries. The following is an example query written by their query language.

- Find the grandchildren elements of this_report.

```
select y
from this_reportPATH_x(y)
where length(PATH_x) = 2
```

In the above example, $PATH\_x$ and y are path variables. The variable $PATH\_x$ ranges through all paths beginning from the root of *this_report*, and y is assigned the elements that satisfy the *length* function in the *where* clause.

Their query language has advantage over previous query languages. First, structure queries can be easily generated by using the path expression. Second, users do not need to know the detailed DTD structure and element names. Third, users feel more friendliness than procedural query languages since their query language is declarative and users do not need to know the detailed procedure of query evaluation. Finally, users can be easily accustomed to their query language because the syntax of their language is similar to the SQL.

However, their language still needs to be improved. First, the path is forward directional only. Thus, the backward path from a low-level node to a higher-level node is not expressible. Second, it is inconvenient to select a group of nodes reachable by any arbitrary number of branches of the tree starting from a node. We need a more powerful path expression which is able to easily control the number of hops during the traversal of the document tree. Third, they have used the same path expression for both document tree and hypertext link structure addressing. That is because they have concentrated on the hypertext structure within a document. However, if we consider general hypertext documents, the hypertext link structure is much different from the hierarchical document structure. That is, the document logical structure is a tree while the hypertext link structure is a directed graph. In order to support general hypertext structure queries, we would need a different path expression and query evaluation scheme for hypertext structure queries. Finally, it is difficult to combine several path expressions together to represent dynamic path changes in the document graph because the path expression itself cannot control the direction and number of hops dynamically. Their query language has been a starting point of our query language design. In Chapter 4, we present a query language which overcomes these shortcomings.

2.5 Structure Representation

Choy et al. [24] have presented a document model and relational database schema for structured documents. In their model, the hierarchical structure of a document is represented as a composite schema. The following is an example of the composite schema for a book.

```
book (title STRING,
      author (name STRING, affiliation STRING),
      publisher STRING,
      date STRING)
```

In the above example, *book* and *author* are composite schemas representing nested data components. A *book* consists of a title, one or more authors, a publisher, and a date. Similarly, an *author* consists of a name and an affiliation. In order to map the composite schema into the database, the schema should be flattened and each composite is stored into a separate relation. That is, the above composite schema is translated into two relations as follows.

```
book = (book-surrogate, title, publisher, date)
author = (author-surrogate, name, affiliation, book-surrogate)
```

The *book-surrogate* and *author-surrogate* are unique identifiers used to access a tuple in the relation. The *book-surrogate* in the relation *author* is a foreign key of the relation *book* and represents the parent-child relationship in the document tree. Using the surrogates, document structure queries can be processed. However, heavy database join operations are required to find a relationship between document elements because the structure information is scattered into several database relations.

Blake et al. [17] have used the parse tree to represent structure relationships between document elements. This approach avoids costly database join operations during structure query processing by maintaining the structure information of a document together in a parse tree. This approach has also been studied in [16] [40] [43] in order to use parse trees as database schemas. However, in this approach, the space overhead is considerable because each document should have an associated parse tree. Moreover, this approach requires much time to access all related parse trees during query processing.

Macleod [59] has represented a document as a tree structure and has suggested an internal tree representation scheme. A document tree is represented as a table with five elements as follows:

- document element identifier
- index number representing the order of the element
- offset (word count) representing the start of the element
- length of the element (number of words)
- offset (word count) of the parent element

The storage requirement of this scheme is considerable. Even though a structure query can be performed using this table, we can use it only if we need to process very detailed level (word level) structure queries with the whole document approach.

The ELF model proposed in [6] does not require database join operations either. In this model, each document element is assigned an EID (absolute element identifier) which is used to access the element. To represent structure information of a document, they have used the LOC (location of element within document) which is similar to the tree location address of the HyQ [55]. During structure query processing, the LOC of each element must be obtained to identify where the element is located within a particular document. However, the problem of this approach is that the identifiers of the ancestors and descendents cannot be obtained directly from the element identifier and LOC of an element. Thus, they have to access the database again to obtain the required information.

2.6 Indexing Structured Documents

Much research has been carried out to design efficient index structures for database and information retrieval systems [11] [19] [30] [33] [34] [35] [36] [41] [47] [67] [68]. Index structures for hypertext systems have been proposed in [19] [20] [30]. These are useful for selecting interested nodes from a hypertext network. However, they cannot be used for structure query processing because they have not considered the logical document structure within a node. To process structure queries, we need an index structure that supports direct access to document elements in the document tree. Recently, Sacks-Davis et al. [66] have proposed some possible inverted index structures for structure query processing.

The first approach is to maintain the inverted index supporting only document access. That is, the inverted list contains only document identifiers, and no element identifiers. If users want to access document elements, document identifiers are first obtained using the document index, and then the elements satisfying the query are identified from the documents. The disadvantage of this approach is that post-processing is required to locate document elements after identifying documents. The cost of the post-processing would be considerable if the number of retrieved documents is large. To reduce the post-processing cost, an internal tree representation [59] can be used. However, the storage overhead is considerable.

The second approach is to maintain separate indices for some selected element types or for all element types. That is, each element type such as section or paragraph will be assigned a separate inverted list. However, the index term should appear in every index, causing considerable storage overhead. For example, an index term "multimedia" is duplicated in all inverted lists. In order to reduce the duplication, a single integrated vocabulary can be used by associating a separate inverted list for each element type with index term. For example, an index term "multimedia" appears in only one combined vocabulary. However, the storage overhead of the inverted list caused by the element identifiers is still considerable. For example, if a paragraph contains a keyword "multimedia," the inverted lists for the keyword "multimedia" will contain all the element identifiers in the path from the root to the paragraph.

Finally, they have proposed a new scheme, called element locator scheme, for document structure indexing. In this approach, each index term in the inverted list is associated with "a path encoding form the root to the leaf and any necessary sibling numbers." This approach also causes considerable duplications in the inverted list.

2.7 Structured Video Documents

Multimedia document management systems such as [14] [25] have been developed. However, they have not considered structure queries. Oomoto and Tanaka [63] have presented a video-object database system named OVID. They list the main features of the system as follows:

- Schemaless description of database
- Interval inclusion hierarchy
- Composition of video objects based on is-a hierarchy

In their system, users define attribute structures for video objects without defining a specific database schema. By using the concept of interval inclusion, annotations of a video object can be inherited to its subobjects. In order to create new video objects, they have defined composition operations such as interval projection, merge, and overlap. A query language named VideoSQL has been used to retrieve video objects based on the content. However, they have not considered the structure query and the index structure supporting the interval inclusion hierarchy.

Weiss et al. [76] [77] have defined an algebraic video data model that has the following characteristics:

- Providing basic operations to create a video stream
- Modeling the hierarchical logical structure of video
- Accessing video by the content and logical/temporal structure

A video segment can be associated with descriptions extracted from text captions or images, and with users' annotations or structural and temporal information between video segments. The descriptions and annotations are used to support content based access to video. Even though they support the query based on the video structure, it is primitive compared to the structure queries of the text document management, and it can be supported only if users associate the structural information to the video segment.

Chapter 3

The Structured Hyperdocument Model (SHM)

3.1 Introduction

Even though the within-component layer of the Dexter model is beyond its scope, the SGML or ODA model can be used in the within-component layer with the Dexter model [7]. The Structured Hyperdocument Model (SHM) defined in this chapter differs from the Dexter model because the SHM can represent the document structure of the within-component layer in order to fully support structure queries. Thus, the goal of the SHM is to represent the document logical structure as well as the hypertext links.

Structured hyperdocuments such as SGML and HTML documents have two kinds of document structures: hierarchical logical structure and hypertext link structure. The logical structure of document elements, such as sections, subsections, and paragraphs, forms a tree. For example, subsections belonging to a section can be considered as child nodes of the section node in the document tree. The hypertext link structure can be represented as a directed graph. A directed edge between nodes in the graph represents a hypertext link. In order to handle structured hyperdocuments using databases, both structures should be supported by the data model on which the query language is based.

In this chapter, we define the SHM which integrates the document logical structure and hypertext link structure. By combining two kinds of document structures, we can build a hyperdocument management system with powerful query facilities.

3.2 The Structured Hyperdocument Model (SHM)

The SHM, represented as a graph, forms a conceptual hyperdocument structure on which a query language is based. Users are able to navigate the hyperdocument graph through queries which exploit the structure. The SHM integrates the logical structure and hypertext structure of documents. The hyperdocument graph is defined as follows:

- The *hyperdocument graph* consists of a set of nodes N and a set of edges E between pairs of nodes.
- A node belonging to N can be either an element of a document such as a subsection or a single element node. Thus, a document consisting of a group of nodes forms a composite node. The data type of a node can be either text or multimedia such as video, audio, image, or animation.

- E has two types of arcs: One represents the hierarchical logical structure (part-of relationship) of a document, and the other represents hypertext links between nodes within a document or between documents. Thus, a composite node can contain hypertext links within itself. The first type of arc forms a document tree, and the second type forms a hypertext network.

Here, we do not restrict the hypertext network to a directed acyclic graph (DAG). It can be a directed cyclic graph. An example of the hyperdocument graph is illustrated in Figure 3.1. It shows two composite nodes, A and B, and four single element nodes, j, k, l, and m. The data type of the node l and m can be multimedia.

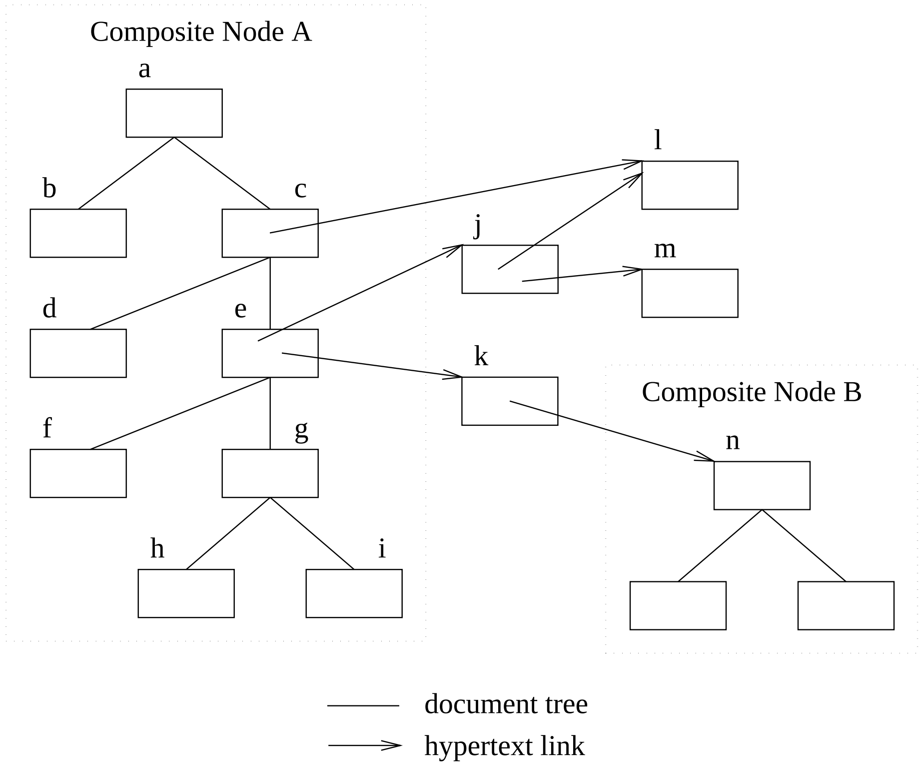

Figure 3.1: Example Hyperdocument Graph

Using the hyperdocument graph, several kinds of queries exploiting the hyperdocument structure can be developed. These queries will be discussed in the next chapter.

3.3 The Extended Structured Hyperdocument Model (ESHM)

In this section, we extend the SHM to support document element sharing and typed hypertext links. The ESHM is also used to support document changes.

3.3.1 Document Element Sharing

In the SHM, we have assumed that a document stored in the database is static and does not change. Thus, it is a static document model. However, in the real world, a document element can be shared by multiple documents. When a document is newly created, existing documents in the database can be reused by allowing document element sharing. Also documents can change by the document manipulation operations such as insertion, deletion, and modification. A new version of a document can be created from an old version of the document and the unchanged elements can be shared by two versions.

In order to support document element sharing, we extend the SHM. In the Extended Structured Hyperdocument Model (ESHM), a document element can be shared by multiple documents. Figure 3.2 shows an example document element which is shared by two documents.

3.3.2 Typed Links

Typed links can be useful in hypermedia systems since they enable users to identify the hypertext links of interest in the hypertext network. Parunak [65] has classi-

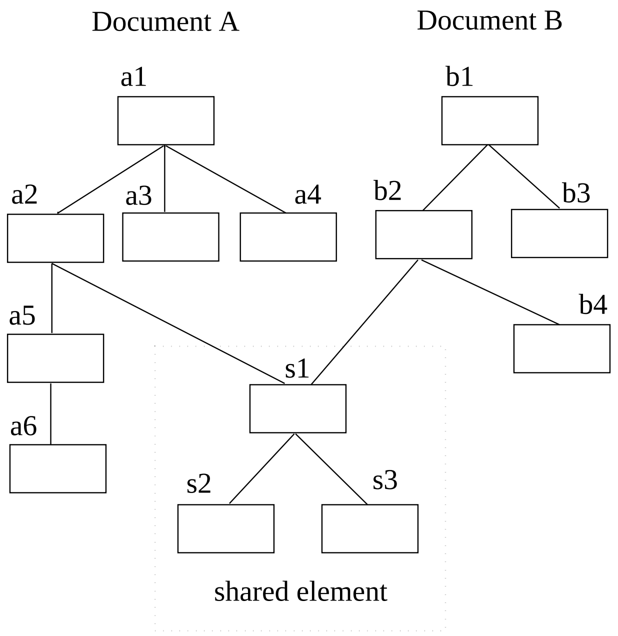

Figure 3.2: Document Element Sharing

fied hypertext links into three categories: association links, aggregation links, and revision links. Each link type is subdivided into subtypes as shown in Table 3.1.

In the SHM, we have not considered typed links. However, in the ESHM, we associate a link in the hyperdocument graph with a type that represents the type of the link in order to support typed links.

3.4 Summary

In this chapter, we have presented the Structured Hyperdocument Model (SHM) which integrates the hierarchical structure and hypertext structure of documents. Whereas the Dexter model considers the node structure of the within-component layer as being outside its scope, our model presents the node structure as well as hypertext links. The hierarchical structure forms a document tree and the hypertext

Table 3.1: Hypermedia Link Types

type	subtype
association link	content link, identification link, comment link, orientation link, implication link, paraphrase link, illustration link
aggregation link	argument aggregate, discussion aggregate, legal case aggregate, software module aggregate
revision link	version link

links form a hypertext network. In order to support document sharing, document changes, and typed links, we have presented the Extended Structured Hyperdocument Model (ESHM).

Chapter 4

The Structured Hyperdocument Query Language (SHQL)

4.1 Introduction

There are four types of queries which should be supported in structured hyperdocument management systems as follows:

- Content query: query based on the document content
- Structure query: query based on the document structure (document tree, hypertext network)
  - Simple structure query: structure query which is not combined with the content query
  - Complex structure query: structure query combined with the content query

- Attribute query: query based on document attributes (document type, security, version)
- Multimedia query: query based on multimedia data

These types can be combined to make a more powerful query.

The Structured Hyperdocument Query Language (SHQL) has a similar format to the SQL relational database query language. A query consists of three clauses: target clause, range clause, and qualification clause as follows:

```
<query> ::=
   find <target clause>
   from <range clause>
   [where <qualification clause>]
```

A detailed description of the SHQL definition appears in the Appendix.

4.2 Content Query

The content query is the most basic query that must be supported for information retrieval applications and is what makes the document database worthy of being built because with it any document can be retrieved based on the content of the document. Content queries can be grouped into two types, which are boolean queries and ranked queries. Boolean queries allow queries only based on the existence of the terms in the documents while ranked queries allow queries based on the importance of the given terms in the documents in some kind of predefined measure.

Sample content queries in the SHQL format are given below. The query examples in this research are similar to those in [6] [17] [27].

- Boolean queries using keywords
  - Find documents related to "structure" and "query."

```
find d
from d:*
where d contains "structure" and "query"
```

- Ranked queries using weighted keywords
  - Find documents about "language" with weight above 0.5.

```
find d
from d:*
where weight(d, "language") > 0.5
```

4.3 Structure Query

It is important to have some queries based on the structure of the hyperdocuments modeled as a hyperdocument graph. Thus, we can ask questions based on the logical structure of the documents such as the chapters, sections, or subsections, etc. Also by utilizing the structure query, we can implement the queries on hypertext links, and navigate through the hypertext network. The structure query can be combined with the content query to build a powerful structure query. In this section, we explain basic concepts to address a document element in a hyperdocument graph and show example structure queries.

In the structure query, users specify required documents using a query expression. The query expression in the SHQL is defined as follows:

- A query 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, link, or list expression.

The branch expression is used to address document trees, the link expression is used to address hypertext links, and the list expression is used to select elements from a list.

4.3.1 Document Tree Addressing

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

- The forward branch expression addresses the subtrees of the current node in the document tree.
  - 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 addresses the ancestor nodes of the current node in the document tree.
  - 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 are the examples of the branch expression for the hyperdocument graph of Figure 3.1.

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

Now we can generate structure queries using the branch expression.

- Find documents that conform to the "report" document type.

```
find r
from r:report
```

- Find books that have more than 10 chapters.

```
find b
from b:book
where count(b.chapter) > 10
```

- Find chapters with sections related to "multimedia" with weight above 0.5.

```
find c
from d:*, c:d.chapter, s:c.section
where weight(s, "multimedia") > 0.5
```

- Find books and reports whose titles contain "multimedia."

```
find d
from d:book or report
where d.title contains "multimedia"
```

- Select all paragraphs containing "New York."

```
find p
from d:*, p:d.paragraph
where p contains "New York"
```

- Find book chapters which have a section containing "hypermedia."

```
find c
from b:book, c:b.chapter
where exists (c.branch[1] contains "hypermedia")
```

- Select all titles within this_report.

```
find t
from r:this_report, t:r.branch[*].title
```

- Select all leaf nodes of this_node.

```
find n.branch[>]
from n:this_node
```

- Select all nodes having a grandchild.

```
find n
from n:node
where exists(n.branch[2])
```

- Select the children and grandchildren nodes of this_node.

```
find e
from n:this_node, e:n.branch[1-2]
```

- Select the siblings and grandchildren nodes of this_node.

```
find difference(n.branch[-1].branch[1], n), n.branch[2]
from n:this_node
```

4.3.2 Hypertext Addressing

Hypertext addressing is a method which makes the hypertext navigation possible. In a simple implementation of a hypertext system, it would be possible that only one level of navigation at a time is supported, so that the user can traverse back and forth only. However, the link expression is designed to give complete addressing capability to the hypertext system, so that the user can navigate to wherever he/she wants as long as there is a route connected by hypertext links. Navigation is always either forward or backward and both types of navigation are accomplished by link expressions.

The link expression is defined similarly as the branch expression.

- The forward link expression is designed to allow forward navigation through the hypertext network.
  - link[*] : nodes reachable by 0 or more forward hops in the hypertext network.
  - 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 designed to allow backward navigation through the hypertext network.
  - link[/] : nodes reachable by 0 or more backward hops in the hypertext network.

– 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 are examples of the link expressions for the hyperdocument graph of Figure 3.1.

```
e.link[*] = {e, j, k, l, m, n}
l.link[/] = {c, e, j, l}
e.link[+] = {j, k, l, m, n}
l.link[-] = {c, e, j}
e.link[1] = {j, k}
l.link[-1] = {c, j}
e.link[2] = {l, m, n}
l.link[-2] = {e}
e.link[0] = {e}
e.link[1-2] = {e.link[1],e.link[2]} = {j, k, l, m, n}
```

The following are query examples using the link expression.

- Find the images connected by a link to a section of the report containing the keyword "hypermedia."

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

- Find the elements containing a link to the second section of this_report.

```
find e
from r:this_report, e:r.element, ss:r.section[2]
where ss in e.link[1]
```

- Find the elements having a link to the last subsection of the first section of this_report.

```
find e
from r:this_report, e:r.element, ss:r.section[1].subsection[>]
where ss in e.link[1]
```

- Select all nodes having a link to this_node.

```
find n.link[-1]
from n:this_node
```

- Select all nodes reachable by a link from this_node.

```
find n.link[1]
from n:this_node
```

- Find the elements referring to an element in the last section of this_report.

```
find e
from r:this_report, e:r.element, ee:r.section[>].element
where ee in e.link[1]
```

- Find the elements in the second chapter of this_book reachable indirectly by two or more hops from an element of the first chapter.

```
find e
from b:this_book, e:b.chapter[2].element,
     ee:b.chapter[1].element
where e in ee.link[2->]
```

4.3.3 List Addressing

List addressing is used to locate a certain portion of the document which forms an element in a document hierarchy such as the first section of a certain chapter, the third author of a certain document, or the last paragraph of a certain section. It uses a technique similar to array addressing to describe the wanted elements of the document hierarchy. The reason that this addressing method is called 'list addressing' is that the subelements are considered as the list of their parent element.

The list expression is defined as follows:

- L[r_1, r_2, ..., r_k] is a list expression, where L is an element name, branch expression, or link expression; $k \geq 1$; r_i is a positive integer, positive integer range, or character '>'. It returns the list of elements selected by the given r_i as follows:
  - If r_i is a positive integer j, the j-th element is selected.
  - If r_i is a positive integer range j_1 - j_2, the elements in the given range are selected.
  - If r_i is a character '>', the last element is selected.

Some examples of the list expression are given below.

```
author[1]  : first author
section[3] : third section
section[>] : last section
chapter[>] : last chapter
section[2].paragraph[>] : last paragraph of the second section
branch[1][2] : second child of a node in the document tree
link[1][1] : first link of a node in the hypertext graph
```

Some query examples using the list expression are given below.

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

```
find r.title, r.author[1]
from r:report
where r.section contains "database" and "hypermedia"
```

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

```
find r.section[2].paragraph[>]
from r:this_report
```

- Find the first section of the report which contains the word "hypermedia."

```
find r.section[1]
from r:report
where r contains "hypermedia"
```

- Find the elements that refer to the second subsection of the first section of this_report.

```
find e
from r:this_report, e:r.element, ss:r.section[1].subsection[2]
where ss in e.link[1]
```

4.3.4 Combination of Addressing Methods

The branch expression, link expression, and list expression can be combined to make more powerful path expressions. Using combined path expressions, users can write complicated queries easily. The following examples show combined expressions for the hyperdocument graph of Figure 3.1.

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

The following are query examples.

- Find the titles of the reports connected to the last child of this_node.

```
find r.title
from r:report, n:this_node
where r in n.branch[1][>].link[1]
```

- Find all multimedia nodes which are connected to this composite node.

```
find m
from m:multimedia, n:this_composite_node
where m in n.branch[*].link[*]
```

4.3.5 Typed Link Expression

In order to handle typed links, the link expression must be able to select some link types during the hypertext navigation. This can be solved by defining the typed link expression.

- The typed link expression has two parameters: link[*parameter*, *type-set*]. The *parameter* is defined as the same as that of the link expression. The *type-set* represents the set of types which will be considered during the hypertext navigation.

For example, if the *type-set* has the value {association, aggregation}, only those two types of links will be followed during the hypertext navigation.

4.4 Attribute Query

Another query type which should be supported is the attribute query which allows users to retrieve a document based on the known properties of the document. Users can attach attributes to a document and use them through the attribute query. The following are the examples of the attribute query usage.

- Find authors of documents with attribute confidential = "yes."

```
find a
from d:*, a:d.author
where d.confidential = "yes"
```

- Find the report whose identifier is "123."

```
find r
from r:report
where r.identifier = "123"
```

- Find the first section of the reports with attribute creation-date = "1996."

```
find r.section[1]
from r:report
where r.creation-date = "1996"
```

4.5 Multimedia Query

Multimedia data are treated the same as text data in our query language. Each multimedia data is stored separately in the database and associated with keywords. Keywords of multimedia data can be extracted from the text descriptions, captions, or annotations of the multimedia data, or can be generated from the data by using a keyword generation tool. Using these keywords, multimedia data can be retrieved by the SHQL query.

- Boolean query using keywords
  - Find images related to "mountain" and "Asia."

```
find d
from d:image
where d contains "mountain" and "Asia"
```

- Ranked query using weighted keywords
  - Find images related to "mountain" with weight above 0.5.

```
find d
from d:image
where weight(d, "mountain") > 0.5
```

- Structure Query
  - Find all nodes which have a link to this_image.

```
find n
from n:node, i:this_image
where i in n.link[1]
```

  - Find all multimedia nodes which are connected to the nodes that have a link to this_image.

```
find m
from m:multimedia, n:node, i:this_image
where (m in n.link[1]) and (i in n.link[1])
```

  - Find the title of the documents which have a link to this_video.

```
find t
from d:*, t:d.title, v:this_video
where v in d.branch[*].link[1]
```

  - Find the abstract of the documents created by the same authors as this_video.

```
find a
from d:*, a:d.abstract, v:this_video
where v.author in d.author
```

- Attribute Query
  - Find images with attribute confidential = "yes."

    ```
    find i
    from i:image
    where i.confidential = "yes"
    ```

  - Find audios with attribute creation-date = "1996."

    ```
    find a
    from a:audio
    where a.creation-date = "1996"
    ```

As defined in Chapter 3, the data type of the node in the hyperdocument graph is multimedia. That is the node can be an image, audio, or video data as well as a text data. The multimedia document such as a video document also has a hierarchical structure. Thus, we can develop structure queries on the multimedia document using the same method as with the text document. This will be discussed in detain in Chapter 9.

4.6 Summary

In this chapter, based on the SHM we have defined the Structured Hyperdocument Query Language (SHQL) which has a similar format to the SQL relational database standard query language. The SHQL supports four types of queries: content query, structure query, attribute query, and multimedia query. We have used path expressions to represent structure queries. The path expression consists of three kinds of expressions: branch expression, link expression, and list expression. The branch expression addresses nodes of the document tree, the link expression addresses nodes of the hypertext network, and the list expression addresses elements of the list. To control the number of hops during the hyperdocument graph traversal, we have introduced the notation similar to the regular expression of automata theory. By using the path expression enhanced with the notation of the regular expression, users can freely navigate the hyperdocument graph and easily control the number of edges to be jumped in the graph.

Chapter 5

Hyperdocument Structure Representation

5.1 Introduction

In order to support structure queries, we must maintain the information about the hyperdocument structure, i.e., parent-child relationships and hypertext links between document elements. The previous approaches discussed in Chapter 2 have used database relations, parse trees, or element locators to represent the hierarchical structure information. However, they require considerable storage overhead and disk access time. In this chapter, we propose a novel scheme which represents the hierarchical structure information using unique element identifiers (UID's) without requiring database relations or parse trees. Our scheme represents the hierarchical structure information using UID's. By letting the UID carry information about the document structure, structure queries can be performed efficiently without any help of other database relations or parse trees.

For the hypertext link representation, we use the join index [73], and this will be discussed in Chapter 6.

5.2 Tree Ordering

The first scheme uses the characteristics of the tree structure. We interpret a document structure as a k-ary tree [32] and assign each element (node) a UID according to the order of the level-order tree traversal. For example, for the document tree of Figure 5.1, we assign UID's as Table 5.1 assuming a 3-ary tree. Because we assume the tree is complete, there are some virtual nodes which do not exist. Using these UID's, we can easily obtain the UID's of the parent and child nodes of a specific node without accessing other data structures unlike the previous approaches.

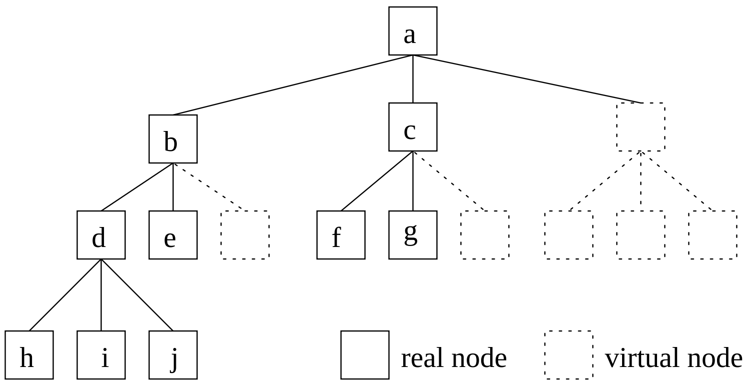

Figure 5.1: Example Document Tree

The UID of the parent node of a node whose UID is i can be obtained by the following function:

$$parent(i) = \left\lfloor \frac{(i-2)}{k} + 1 \right\rfloor . \tag{5.1}$$

The UID of the j-th child node of a node whose UID is i can be obtained by

Table 5.1: UID's for the Tree Ordering Scheme

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

the following function:

$$child(i, j) = k(i - 1) + j + 1. \tag{5.2}$$

When the degree (k) of the tree is the power of 2, the algorithm is much simpler by interpreting the UID's as binary numbers. Document types may provide some guidance in selecting the degree (k) of the tree.

5.2.1 Handling Unbalanced Documents

One possible problem of this scheme is that the document tree may be unbalanced, that is, a specific level of the tree has many more children nodes than other levels. In this case, we need other identifier coding schemes. For example, if the width of the tree is much larger than the depth, we can use the pair (*level number*, *unique identifier in the level*) as the UID. This scheme can represent more document elements since the right-most virtual nodes of each level will not be assigned UID's. The algorithm is similar to the basic tree ordering scheme.

The UID of the parent node of a node at level l whose UID is i can be obtained by the following function:

$$parent(l, i) = \left(l - 1, \left\lfloor \frac{(i-2)}{k} + 1 \right\rfloor\right). \tag{5.3}$$

The UID of the j-th child node of a node at level l whose UID is i can be calculated by the following function:

$$child(l, i, j) = (l + 1, k(i - 1) + j + 1). \tag{5.4}$$

Another coding scheme can also be used. If a document has many children at some specific level, we can introduce a level of virtual nodes to balance the number of children of each level. However, the algorithm is more complicated.

5.3 Tree Locator

The second scheme uses the tree locator which is similar to the tree location address of the HyQ [55] as the UID. In the tree location address, a node address is the concatenation of the children identifiers in the path from the root to the node. For example, the tree location address of the node which is the fourth child of the third child of the second child of the root is '1 2 3 4'. In our scheme, we use '2 3 4' as the

Table 5.2: UID's for the Tree Locator Scheme

element	unique identifier
a	0 0 0
b	0 0 1
c	0 0 2
d	0 1 1
e	0 1 2
f	0 2 1
g	0 2 2
h	1 1 1
i	1 1 2
j	1 1 3

UID of the node and do not include the root identifier which is always '1'. Table 5.2 shows the UID's for Figure 5.1.

The UID of the parent node of a node whose UID is i can be obtained by shifting one digit right as follows:

$$parent(i) = shift\_right(i). \tag{5.5}$$

The UID of the j-th child node of a node whose UID is i can be obtained by shifting one digit left after concatenating i with j as follows:

$$child(i, j) = shift\_left(concatenate(i, j)). \tag{5.6}$$

5.4 Two Level UID

By using the Extended Structured Hyperdocument Model (ESHM) and changing the basic UID scheme, we can support dynamic documents that change over time. In this section, we present a new UID scheme in order to support document sharing and document changes.

5.4.1 Document Sharing

Figure 5.2 is an example document tree which shows an element (s1) is shared by two documents.

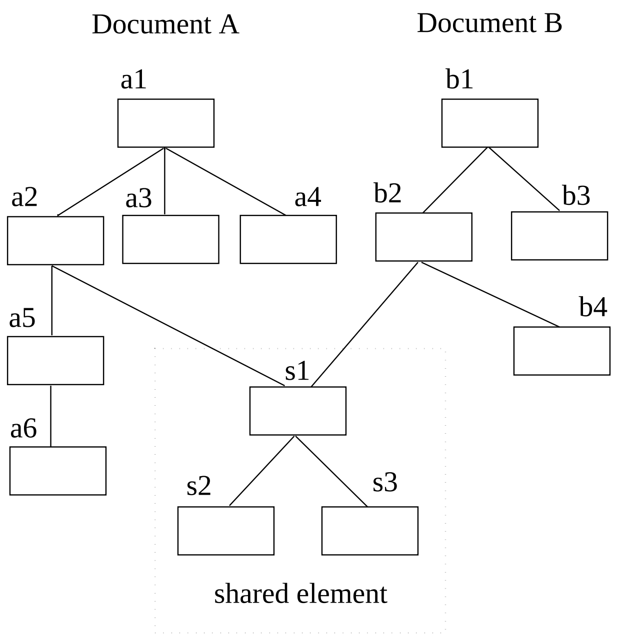

Figure 5.2: Document Sharing

The basic UID coding scheme cannot be used for this type of document because an element belongs to more than one document. We can support it by introducing two level unique element identifiers: logical element identifier and physical element identifier. The logical UID is assigned by using the basic UID scheme and is used for document element indexing and query processing. The logical UID is unique within a document. The physical UID is the one which is used for database access. That is, we access a document element using the physical UID as a key in the database. For the static documents, the logical UID and the physical UID are the same.

Table 5.3 illustrates the logical UID's for Document A of Figure 5.2, and Table 5.4 shows the logical UID's for Document B assuming a 3-ary tree.

Table 5.3: Logical UID's for Document A

element	unique identifier
a1	1
a2	2
a3	3
a4	4
a5	5
a6	14
s1	6
s2	17
s3	18

In order to map the logical UID to the physical UID, we need an element identifier mapping table as shown in Table 5.5. We assume new elements are assigned the physical UID's starting from 1000.

Table 5.6 is the UID mapping table for Document B.

Table 5.4: Logical UID's for Document B

element	unique identifier
b1	1
b2	2
b3	3
b4	6
s1	5
s2	14
s3	15

Table 5.5: UID Mapping Table for Document A

Logical UID	Physical UID
1	1
2	2
3	3
4	4
5	5
14	14
6	1000
17	1001
18	1002

Table 5.6: UID Mapping Table for Document B

Logical UID	Physical UID
1	1
2	2
3	3
6	6
5	1000
14	1001
15	1002

5.4.2 Document Changes

The Two Level UID scheme can be used for supporting document changes such as element insertion and deletion. If we use the basic UID scheme for dynamic documents which change frequently, the UID's of the elements must be changed accordingly. However, by using the Two Level UID scheme, we can minimize database changes.

5.4.2.1 Element Insertion

Figure 5.3 shows that an element (a6) is inserted into a document tree.

Table 5.7 illustrates the logical UID's before insertion, and Table 5.8 shows the logical UID's after insertion. Table 5.9 is the UID mapping table for the document.

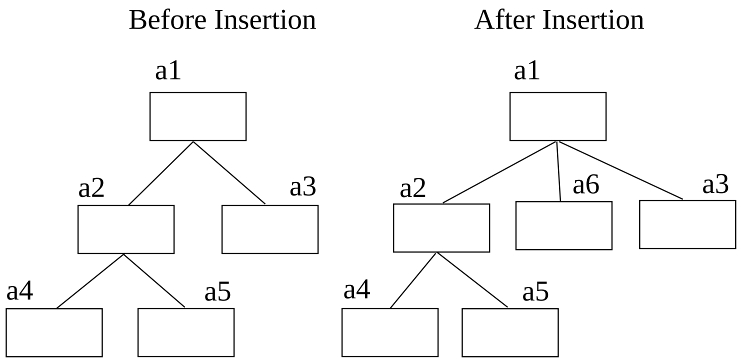

Figure 5.3: Element Insertion

Table 5.7: Logical UID's Before Insertion

element	unique identifier
a1	1
a2	2
a3	3
a4	5
a5	6

Table 5.8: Logical UID's After Insertion

element	unique identifier
a1	1
a2	2
a6	3
a3	4
a4	5
a5	6

Table 5.9: UID Mapping Table After Insertion

Logical UID	Physical UID
1	1
2	2
3	1000
4	3
5	5
6	6

5.4.2.2 Element Deletion

We can use the same scheme as with the case of element insertion. Figure 5.4 shows that an element (a3) is deleted from a document tree.

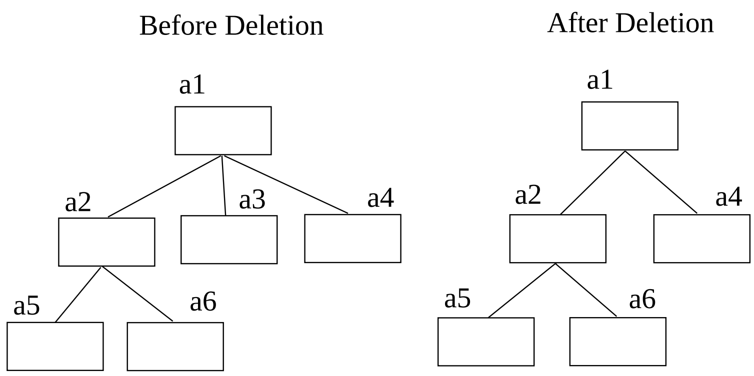

Figure 5.4: Element Deletion

Table 5.10 shows the logical UID's before deletion and Table 5.11 illustrates the logical UID's after deletion assuming a 3-ary tree. Table 5.12 is the UID mapping table for the document.

Table 5.10: Logical UID's Before Deletion

element	unique identifier
a1	1
a2	2
a3	3
a4	4
a5	5
a6	6

Table 5.11: Logical UID's After Deletion

element	unique identifier
a1	1
a2	2
a4	3
a5	5
a6	6

Table 5.12: UID Mapping Table After Deletion

Logical UID	Physical UID
1	1
2	2
3	4
5	5
6	6

5.5 Summary

In order to represent the hierarchical document structure, we have proposed two kinds of new document element identifier (UID) coding schemes: the tree ordering scheme and the tree locator scheme. In both schemes, the UID has been designed to carry the information about the document structure. By using the UID, the UID's of the ancestors or descendents of a node can be obtained directly using the *parent* and *child* function. Thus, our scheme does not require other data structures such as database relations or parse trees to store the structure information. Our scheme will be used in the later chapters to design efficient index structures and to process structure queries efficiently. In order to support document sharing and document changes, we have proposed the Two Level UID scheme.

Chapter 6

Indexing Structured Hyperdocuments

6.1 Introduction

Much research has been done in order to design efficient index structures for database, hypermedia, and information retrieval systems [11] [19] [30] [34] [35] [47] [67] [68]. To process structure queries, we need an index structure that supports direct access to document elements. However, there has been little research on the index structures for structured documents. If we use the conventional indexing techniques for document element indexing, there is considerable storage and access time overhead for structure query processing. In this chapter, we propose new index structures based on the inverted index and signature file that facilitate structure query processing.

An index structure for hypertext links is also needed to evaluate link expressions. For the fast hypertext link traversal, we use the join index proposed by Valduriez [73].

We define index expressions in order to easily describe the index structures. The syntax is similar to the path expression.

- INDEX[+] : indices of all nodes in the subtrees of the current node in the document tree.
- INDEX[-] : indices of all the ancestor nodes of the current node in the document tree.
- INDEX[0] : index of the current node.
- INDEX[>] : indices of all the leaf nodes in the subtrees of the current node.

6.2 Inverted Index

Since users can access any kind of document element in the document tree, it is necessary to build an inverted index structure that facilitates access to an element in the database. We propose various kinds of new inverted index structures which support fast access to document elements.

In order to support direct element access, it is required to include all the index terms of elements in the inverted list. In the document tree, whereas the leaf nodes are associated with data, the internal nodes represent only structure relationships between document elements. Suppose that a document tree has three leaf nodes and each leaf element has index terms as shown in Figure 6.1. Here, we can consider the node A is a chapter, B and C are sections, and D and E are subsections.

Even though the internal nodes have no associated document data, the data at the subtrees must be considered as their data. Thus, the index terms for each node are as follows:

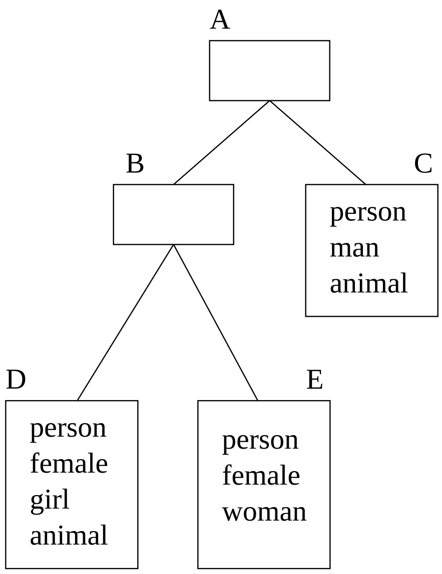

Figure 6.1: Document Tree with Index Terms

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

Now the problem is how to maintain an inverted index structure for fast access to the document elements.

6.2.1 Inverted Index for All Nodes with Replication (ANWR)

The first naive approach is to replicate all index terms of the children to their ancestor nodes as shown in Figure 6.2.

This scheme is based on the conventional inverted index, and the inverted list

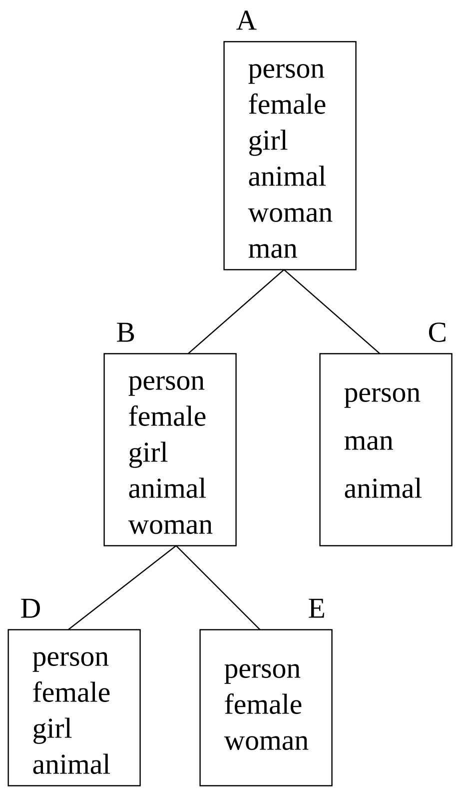

Figure 6.2: Indices for All Nodes with Replication

for an index term must include all the UID's of the elements which contain it. Thus, the index of a specific node is INDEX[0]. The inverted list for the example is as follows:

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

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.

6.2.2 Inverted Index for All Levels with Replication (ALWR)

This scheme is the same as the ANWR except that each element type has a separate inverted list per index term. This scheme has also been described in [66] without analysis. The inverted list for the example is as follows:

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

The inverted list for "person" has three sub-lists for three element types. This scheme also has considerable storage overhead in the inverted list.

6.2.3 Inverted Index for Leaf Nodes Only (LNON)

In this scheme, we include element identifiers in the inverted list for leaf nodes only as shown in Figure 6.3.

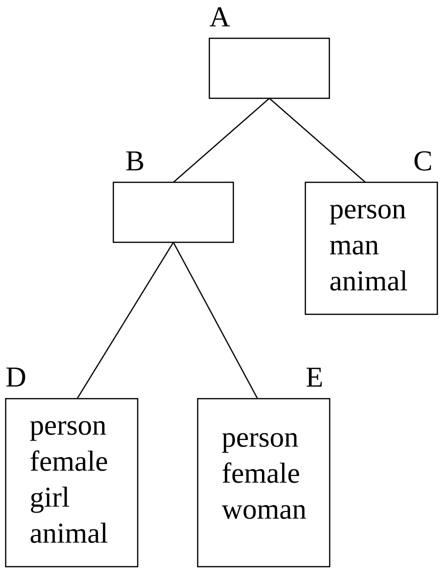

Figure 6.3: Indices for Leaf Nodes Only

In this scheme, the index of a node is INDEX[>]. The inverted list for the example is as follows:

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

```
inverted-list(man) = {C}
```

The storage requirement of this scheme is less than that of the ANWR. Even though we do not maintain the UID's of the internal nodes in the document tree, the UID's of the ancestor nodes of a node can be calculated by the *parent* function defined in Chapter 5.

6.2.4 Inverted Index for All Nodes without Replication (ANOR)

We propose a novel method to build the inverted list which saves more space. Figure 6.4 shows how to build an index for structured documents. In this scheme, we use the fact that the child nodes of a node can have some index terms in common.

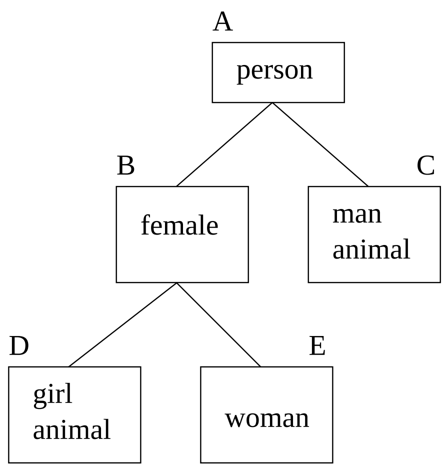

Figure 6.4: Indices without Replication

By using this fact, we can construct the inverted list for the example as follows:

```
inverted-list(person) = {A}
inverted-list(female) = {B}
```

```
inverted-list(girl) = {D}
inverted-list(animal) = {C, D}
inverted-list(woman) = {E}
inverted-list(man) = {C}
```

In this scheme, the index of a node is

$$\text{INDEX[0]} \cup \text{INDEX[+]} \cup \text{INDEX[-]}.$$

That is, the index terms of the internal nodes are inherited to their descendent nodes, and those of the leaf nodes are inherited to their ancestor nodes.

The algorithm to build the inverted list for this scheme is given below.

```
Algorithm: BuildIndex
  1. Traverse the document tree in post order.
  2. Build index for each internal node as follows:
        index(internal node) =
              set-intersection(indices of all child nodes).
  3. Delete the selected index terms from the child nodes.
```

Using the inverted list constructed by the above algorithm, we can access any element in the database using the *parent* and *child* function defined in Chapter 5.

6.2.5 Inverted Index for Root Node Only (RNON)

We can consider an extreme case which has been used in the traditional information retrieval systems. In this scheme, we include only the element identifier of the root node in the inverted list, that is, the document identifier only, as shown in Figure 6.5.

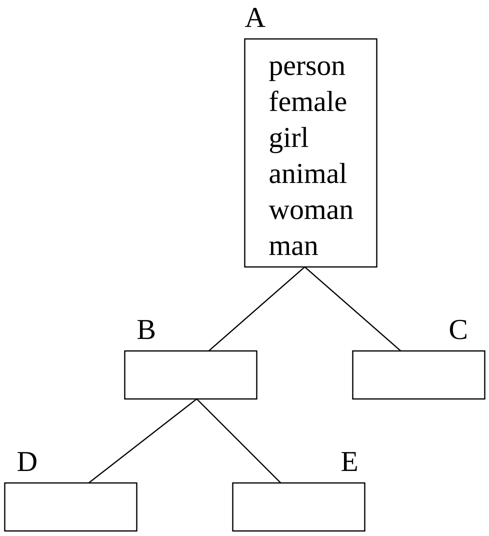

Figure 6.5: Indices for Root Node Only

The inverted list for the example is given below.

```
inverted-list(person) = {A}
inverted-list(female) = {A}
inverted-list(girl) = {A}
inverted-list(animal) = {A}
inverted-list(woman) = {A}
inverted-list(man) = {A}
```

However, this approach is inappropriate for structure query processing. That is, to process structure queries, we have to either access all related documents as a whole or maintain other data structures to identify satisfying element identifiers.

6.3 Signature File

The signature file is a useful indexing technique in information retrieval because its storage utilization is better than that of the inverted index even though it requires more disk access time [34]. In this approach, each document is assigned a signature which is obtained by bitwise ORing keyword signatures produced by applying a hash function to the keywords in the document. Searching for a keyword is handled by obtaining a keyword signature and by probing each document signature for '1' in those bit positions that the keyword signature contains a '1'. In order to process structure queries efficiently, we need a signature file that supports fast element access.

6.3.1 Signature File for All Nodes (SFAN)

In this scheme, we maintain signatures of all elements in the document tree. All signatures of the database are stored into a single signature file. That is, the signature file for Figure 6.1 is composed as follows:

```
signature-file = {signature(A), signature(B), signature(C),
                  signature(D), signature(E)}
```

However, it requires considerable space to maintain element signatures. Moreover, it is inefficient as the entire signature file must be accessed to process a query.

6.3.2 Signature File for All Levels (SFAL)

In this scheme, all element signatures of the document tree are maintained. However, unlike the SFAN, each element type is assigned a separate signature file in order to reduce the time needed to scan the signature file. That is, three signature files are required for Figure 6.1 as follows:

```
signature-file(1) = {signature(A)}
signature-file(2) = {signature(B), signature(C)}
signature-file(3) = {signature(D), signature(E)}
```

Even though this scheme can access document elements very fast, it requires considerable space.

6.3.3 Signature File for Leaf Nodes Only (SFLN)

To reduce the space for the signature file, we maintain signatures of the leaf nodes only and do not maintain signatures of the internal nodes. This scheme is similar to the LNON inverted indexing scheme. The signature file for Figure 6.1 is as follows:

```
signature-file = {signature(C), signature(D), signature(E)}
```

Even though this scheme does not have the signatures of the internal nodes, they can be calculated by using the following function.

```
signature(internal node) = bitwise-OR (signatures of leaf nodes)
```

6.3.4 Signature File for Root Node Only (SFRN)

This scheme is the one used in traditional information retrieval systems. In this scheme, only the root node of the document tree has a signature and the other nodes do not have signatures. The signature file for Figure 6.1 is as follows:

```
signature-file = {signature(A)}
```

However, it is difficult to support structure queries in this scheme.

6.3.5 Signature File for Selected Levels (SFSL)

In order to reduce the time required to perform the bitwise OR operation in the SFLN, we can maintain signatures at some selected levels of the document tree. The SFLN and SFRN are the extreme cases of this scheme. As an example of this scheme, we can combine the SFLN and SFRN schemes, that is, we can maintain signatures for the root node and leaf nodes. We can also maintain three levels of signatures, for example, for the root node, leaf nodes, and middle level nodes.

6.4 Indexing Hypertext Links

In order to support fast forward and backward hypertext link traversal, we need an access structure for hypertext links. In this research, we use the join index [73] to represent hypertext links in the database. The join index consists of six attributes: source document identifier, source element identifier, source anchor address, destination document identifier, destination element identifier, and destination anchor address. One B^+ tree is constructed for the fast access to the pair of the source document identifier and source element identifier, and another B^+ tree is built for the pair of the destination document identifier and destination element identifier.

6.5 Cost Comparison

In this section, we analyze the storage requirements and disk access times of the proposed inverted index structures and signature file structures.

6.5.1 Inverted Index

We analyze the storage requirements and disk access times of the ANWR, ALWR, LNON, and ANOR inverted indexing schemes. The inverted index model which will be used for the analysis is shown in Figure 6.6.

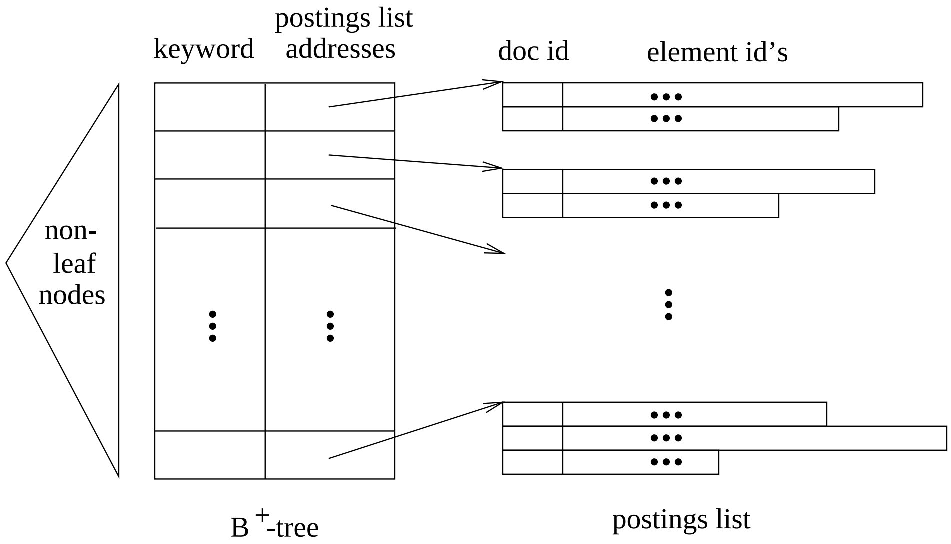

Figure 6.6: Inverted Index Structure

The symbols of Table 6.1 are used in the space and time expressions representing the storage requirements and disk access times.

6.5.1.1 Storage Requirement

The storage requirement of an index structure is the sum of those required for the B^+-tree and postings list.

$$S_{index} = S_{btree} + S_{posting}. \tag{6.1}$$

The B^+-tree has the leaf and internal nodes.

$$S_{btree} = S_{leaf} + S_{internal}. \tag{6.2}$$

Since the total number of keywords in the database is n_{key}, the number of disk blocks required for the leaf nodes is

$$S_{leaf} = \left\lceil \frac{(s_{key} + n_{invert\_list} * s_{ptr}) * n_{key}}{s_{block}} \right\rceil. \tag{6.3}$$

If we assume that the height of the B^+-tree is l and the degree (fan-out) is d ($d > 1$), then the number of disk blocks for the internal nodes is

$$S_{internal} = \sum_{i=0}^{l-2} d^i$$

$$= \frac{d^{l-1} - 1}{d - 1}. \tag{6.4}$$

Table 6.1: Symbols and Definitions for Inverted Index

symbol	definition
d	degree of B^+-tree
h	height of document tree
k	degree of document tree
l	height of B^+-tree
m	average number of keywords in a leaf node
n_{doc}	total number of documents in database
n_{invert_list}	number of inverted lists per index term
n_{key}	total number of keywords in database
p	rate of promoted keywords from children to parent node
s_{block}	block size in bytes
s_{doc_id}	document identifier size in bytes
s_{elem_id}	element identifier size in bytes
s_{key}	average keyword size in bytes
s_{ptr}	pointer size in bytes
t_{random}	random disk block access time in milliseconds
t_{seq}	sequential disk block access time in milliseconds
u	rate of unique index terms in children nodes
$N_{block\_per\_key}$	average number of blocks of postings list per keyword
$N_{doc\_per\_key}$	average number of documents per keyword
$N_{post\_per\_doc}$	average number of postings per document
$N_{post\_per\_key}$	average number of postings per keyword

The degree d can be calculated as follows:

$$d = \left\lfloor \frac{s_{block}}{s_{key} + s_{ptr}} \right\rfloor. \tag{6.5}$$

The storage requirement for the postings list for the database is

$$S_{posting} = \left\lceil \frac{n_{key} * (N_{doc\_per\_key} * s_{doc\_id} + N_{post\_per\_key} * s_{elem\_id})}{s_{block}} \right\rceil. \tag{6.6}$$

The number of postings per keyword is

$$N_{post\_per\_key} = \left\lceil \frac{N_{post\_per\_doc} * n_{doc}}{n_{key}} \right\rceil. \tag{6.7}$$

Now we calculate $N_{doc\_per\_key}$, the average number of documents per keyword. We assume that the document tree is full k-ary and u is the average rate of unique index terms among the child nodes which will be indexed in the parent node. So, $1 - u$ represents the rate of replicated index terms in the child nodes that will not be indexed in the parent node. The range of u is

$$\frac{1}{k} \leq u \leq 1.$$

If u is close to $\frac{1}{k}$, the document is very dense, that is, the elements of the document are closely related to each other and have very similar index terms. When u is large, the document is sparse, that is, the elements of the document are not closely related.

Assume that the average number of index terms of a leaf node is m. Then the number of index terms of a node at level (h - 1) is

$$k * m * u.$$

The number of index terms of a node at level i is

$$k^{h-i} * m * u^{h-i}.$$

Since the number of nodes at level i is k^{i-1}, the total number of index terms at level i is

$$k^{i-1} * k^{h-i} * m * u^{h-i}$$

$$= k^{h-1} * m * u^{h-i}.$$

Since the number of index terms of a root node is

$$k^{h-1} * m * u^{h-1},$$

the number of documents per keyword is

$$N_{doc\_per\_key} = \left\lceil \frac{k^{h-1} * m * u^{h-1} * n_{doc}}{n_{key}} \right\rceil. \tag{6.8}$$

Now we calculate $N_{post\_per\_doc}$, the number of index terms per document, for each inverted indexing scheme.

ANWR:

The total number of index terms per document is

$$N_{post\_per\_doc} = \sum_{i=1}^{h} k^{h-1} * m * u^{h-i}$$

$$= k^{h-1} * m * \sum_{j=0}^{h-1} u^j$$

$$= k^{h-1} * m + k^{h-1} * m * \sum_{j=1}^{h-1} u^j$$

$$= \begin{cases} k^{h-1} * m + k^{h-1} * m * \frac{u*(1-u^{h-1})}{1-u} & \text{if } u \neq 1 \\ k^{h-1} * m * h & \text{if } u = 1 \end{cases} \tag{6.9}$$

ALWR:

For the ALWR, we have to maintain as many inverted lists as the height of the document tree. The total number of index terms per document for the inverted list of level i is the same as the number of index terms at level i. Thus,

$$N_{post\_per\_doc} = k^{h-1} * m * u^{h-i}. \tag{6.10}$$

LNON:

The total number of index terms per document is

$$N_{post\_per\_doc} = k^{h-1} * m \tag{6.11}$$

since the number of leaf nodes in the tree is k^{h-1}.

ANOR:

For the analysis of the ANOR, we introduce a parameter p which represents the probability that an index term is promoted from a child node to its parent node. The range of p is

$$0 \leq p \leq 1.$$

Then, the number of index terms of a leaf is

$$m - m * p.$$

The number of index terms of a node at level i is

$$m * p^{h-i} - m * p^{h-i+1}.$$

The total number of index terms at level i is

$$k^{i-1} * m * p^{h-i} - k^{i-1} * m * p^{h-i+1}$$

because the number of nodes at level i is k^{i-1}. And the number of index terms of the root node is $m * p^{h-1}$. Thus, the total number of index terms of a document is

$$N_{post\_per\_doc} = \sum_{i=2}^{h}(k^{i-1} * m * p^{h-i} - k^{i-1} * m * p^{h-i+1}) + m * p^{h-1}$$

$$= k^{h-1} * m - m * \sum_{i=1}^{h-1} p^{i} * \left(k^{h-i} - k^{h-i-1}\right). \tag{6.12}$$

6.5.1.2 Disk Access Time

The disk access time required to retrieve the postings list of a keyword is the sum of the B^+-tree access time and postings list access time.

$$T_{search} = T_{btree} + T_{posting}. \tag{6.13}$$

The disk access time of the B^+-tree is

$$T_{btree} = l * t_{random}. \tag{6.14}$$

The disk access time of the postings list of a keyword is

$$T_{posting} = t_{random} + (N_{block\_per\_key} - 1) * t_{seq}, \tag{6.15}$$

where $N_{block\_per\_key}$, the number of disk blocks of the postings list per keyword, is

$$N_{block\_per\_key} = \left\lceil \frac{S_{posting}}{n_{key}} \right\rceil. \tag{6.16}$$

6.5.1.3 Analytical Results

We compare the storage requirements of the inverted indexing schemes by using the parameters as shown in Table 6.2. The disk access times are based on the Conner CP30200 model [31].

We assume that the database has 100,000 documents and 50,000 keywords. Figure 6.7 presents the storage requirements of the ANWR, ALWR, LNON, and

Table 6.2: Parameter Settings for Inverted Index

symbol	value
d	64
h	5
k	5
l	4
m	10
n_{doc}	100,000
n_{invert_list}	5 (ALWR), 1 (others)
n_{key}	50,000
s_{block}	1024
s_{doc_id}	4
s_{elem_id}	2
s_{key}	12
s_{ptr}	4
t_{random}	19.06
t_{seq}	0.67
u	varied (0.5, 0.6)

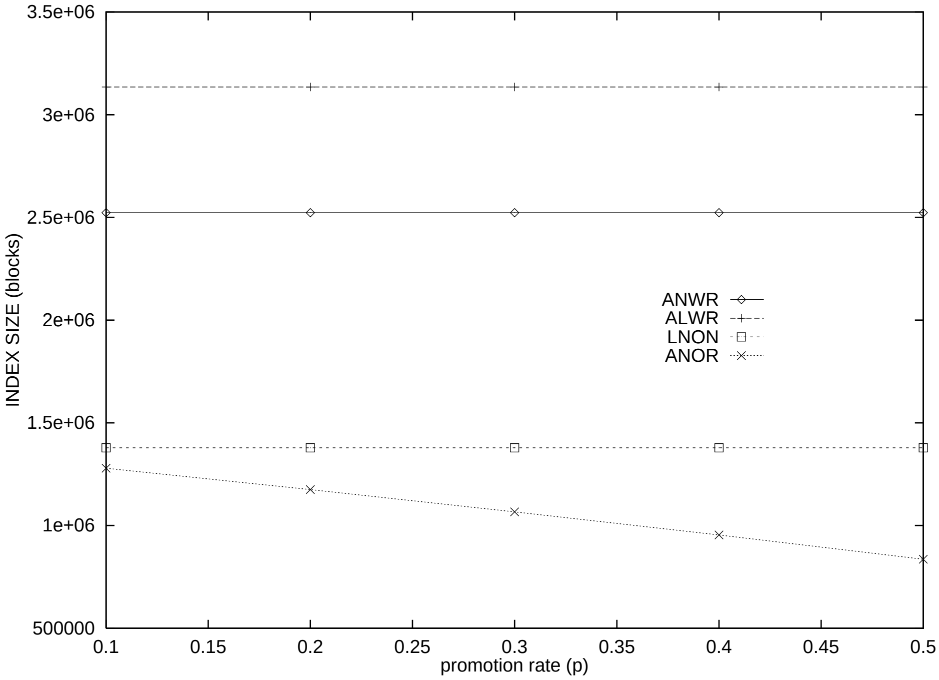

Figure 6.7: Index Block Requirement (u=0.5)

ANOR, when u is 0.5. It illustrates that the ANOR has the best performance in the storage requirement while the ALWR has the worst performance. It also shows that the index size of the ANOR decreases linearly as the value of p increases.

Figure 6.8 illustrates the difference of the storage requirements of the ANWR, ALWR, LNON, and ANOR, when u is 0.6. It shows the ANOR has the least storage requirement.

Figure 6.9 shows the disk access times of the ANWR, ALWR, LNON, and ANOR, when u is 0.5 and a keyword in the leaf node is accessed. The ANOR has the best performance in the disk access time while the ANWR has the worst. The ALWR and LNON has the same disk access time as presented as the same curve in the graph. The disk access time of the ANOR decreases as the value of p increases.

Figure 6.10 illustrates the difference of disk access times of the four schemes

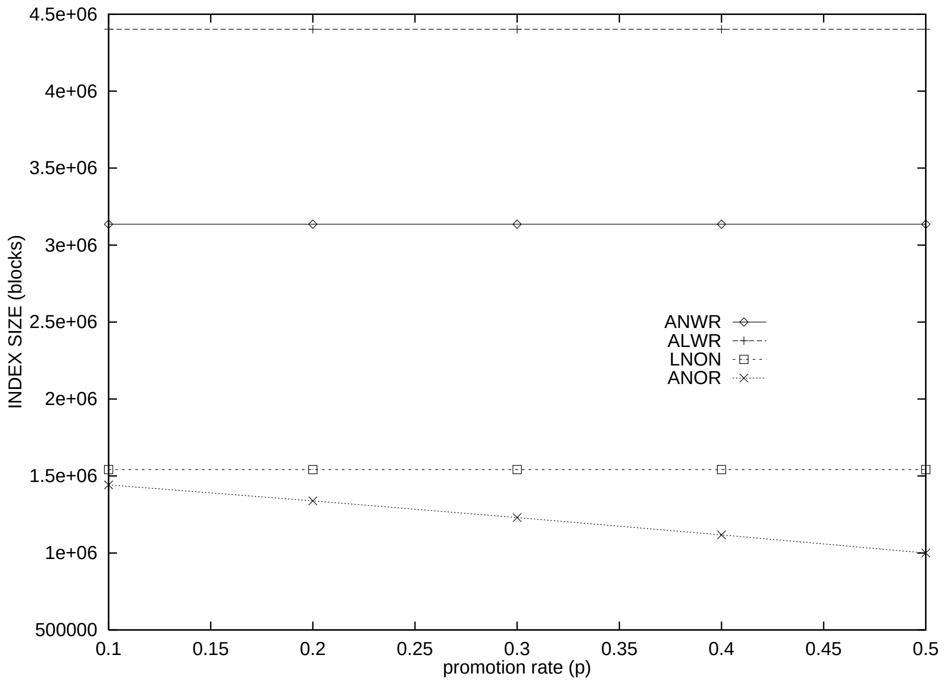

Figure 6.8: Index Block Requirement (u=0.6)

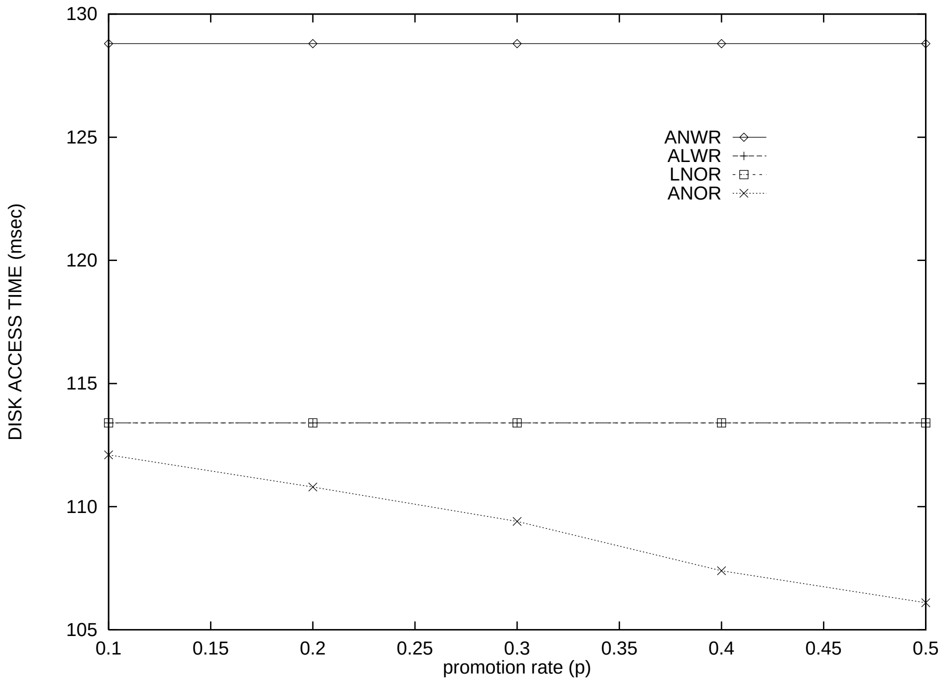

Figure 6.9: Disk Access Time for Leaf Node (u=0.5)

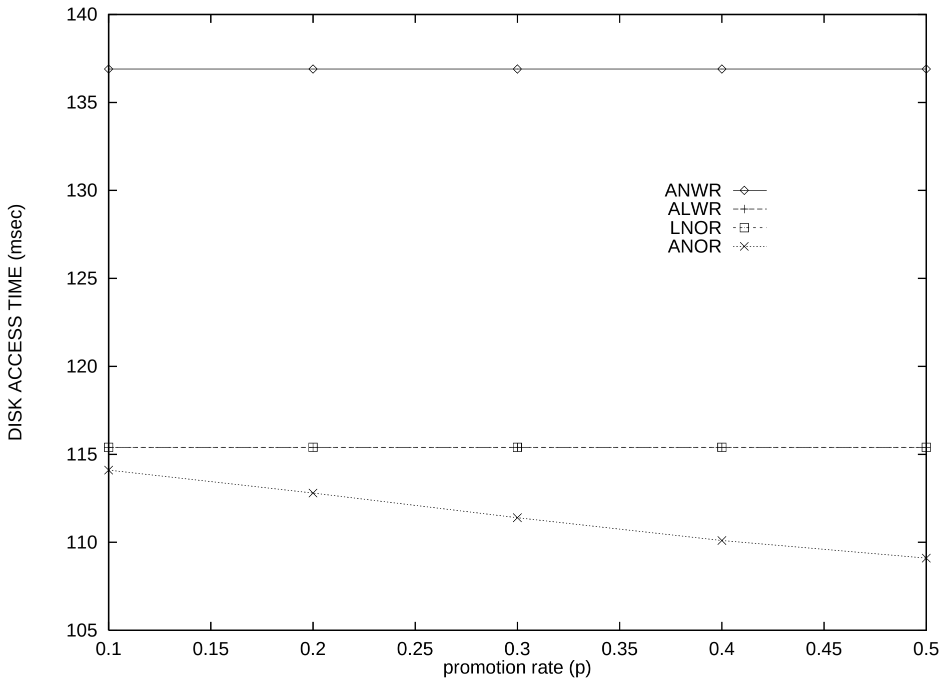

Figure 6.10: Disk Access Time for Leaf Node (u=0.6)

when u is 0.6 and a keyword in the leaf node is accessed.. It also shows that the ANOR has the best performance in the disk access time while the ANWR has the worst performance. The disk access times of the ALWR and LNON are the same and presented as the same curve in the graph.

Figure 6.11 presents the average disk access times of the ANWR, ALWR, LNON, and ANOR, when u is 0.5 and any arbitrary keyword in the document tree is accessed. It shows that the ANOR has the best performance in the average disk access time when p is larger than 0.1.

Figure 6.12 illustrates the difference of disk access times of the four schemes when u is 0.6 and any arbitrary keyword in the document tree is accessed. The ANOR has the best performance in the average disk access time when p is larger than 0.1. We also have obtained similar results with other values of u.

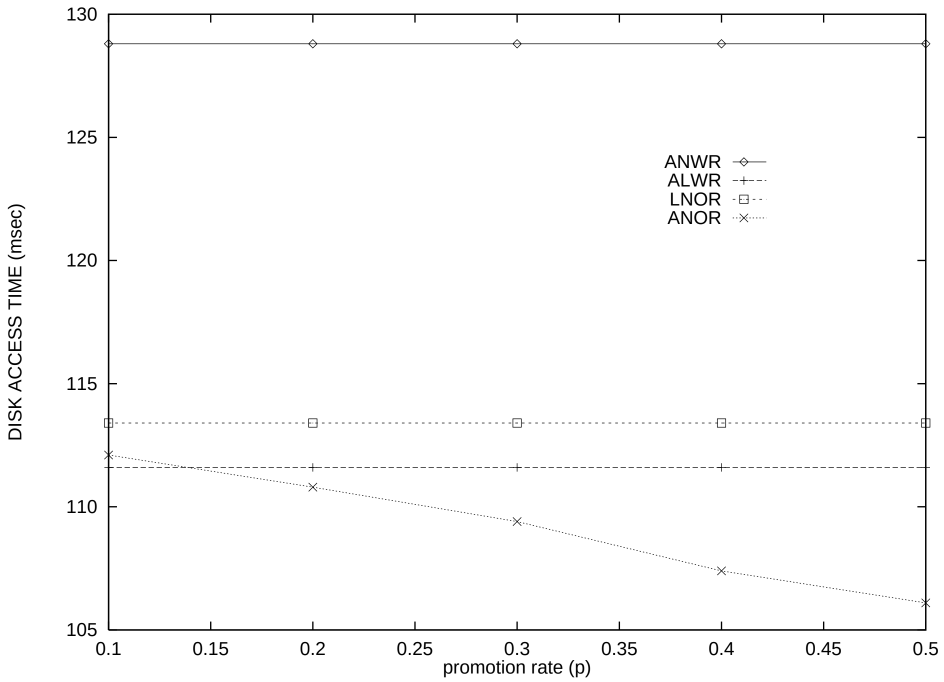

Figure 6.11: Average Disk Access Time (u=0.5)

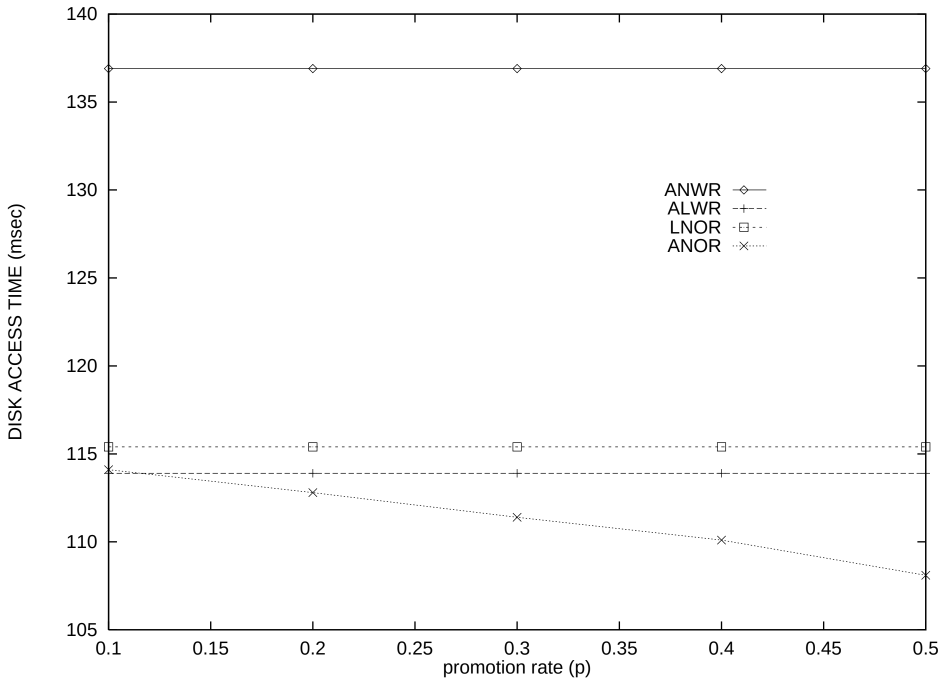

Figure 6.12: Average Disk Access Time (u=0.6)

6.5.1.4 Experimental Results

Experiments have been performed on a SUN SPARC station with a local Conner CP30200 disk. We have experimented in a single user environment after building a partial inverted index in a 100 Mbyte disk space. We have assumed the database contains 100,000 documents with 50,000 keywords.

Figure 6.13 shows the results of the experiment obtained by accessing the index for a given keyword 100 times each and averaging the access times when u is 0.5.

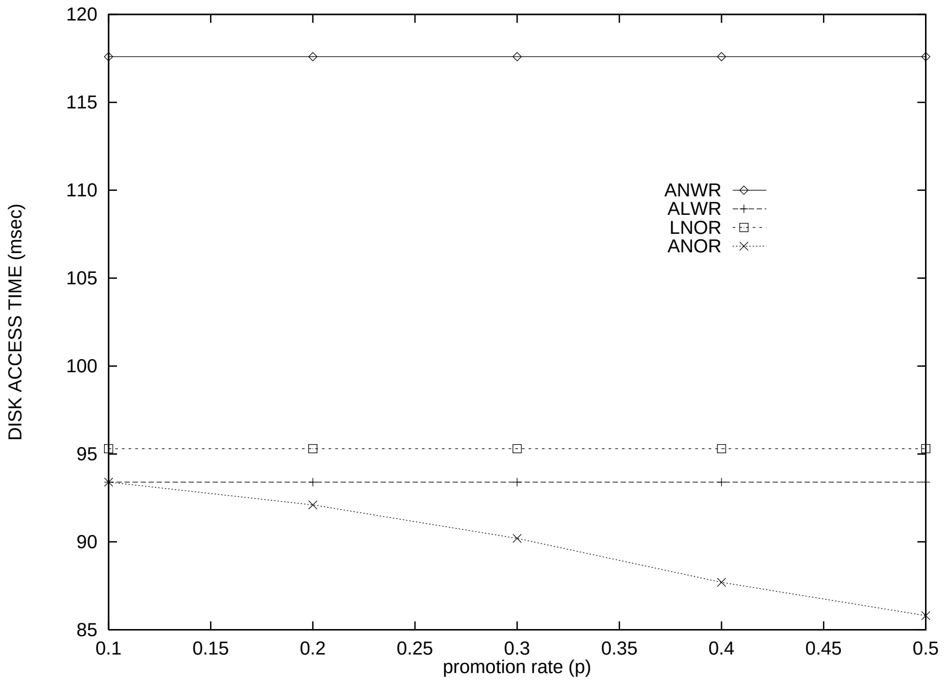

Figure 6.13: Experimental Average Disk Access Time (u=0.5)

The reason that the experimental results are somewhat faster is that we have experimented in a single user environment with a relatively small disk space and we have not considered other aspects such as buffer management in the analysis.

Figure 6.14 presents the experimental results when u is 0.6. The results are similar to the analytical results.

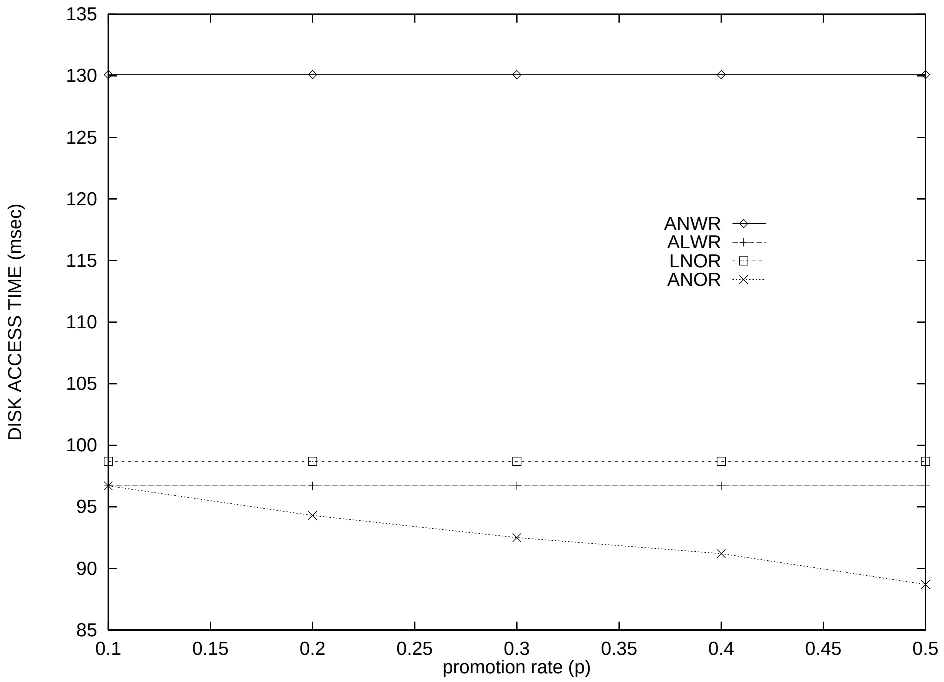

Figure 6.14: Experimental Average Disk Access Time (u=0.6)

6.5.2 Signature File

We analyze the storage requirements and disk access times of the SFAL and SFLN signature file schemes. The signature file model shown in Figure 6.15 is based on the frame sliced signature file [34].

The symbols of Table 6.3 are used in the space and time expressions representing the storage requirements and disk access times.

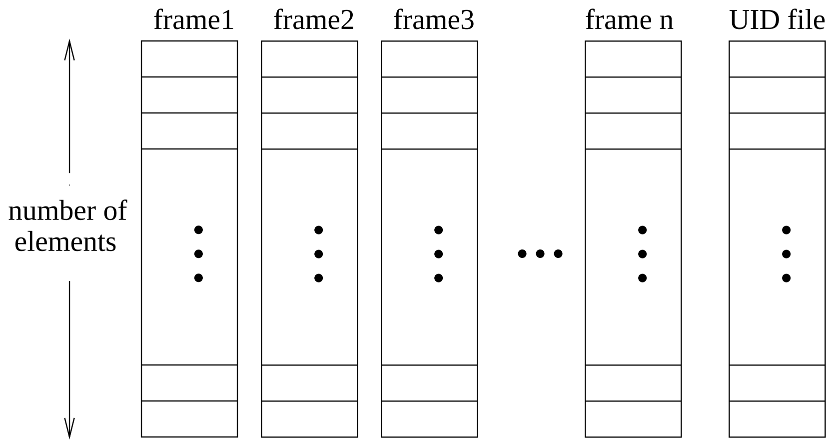

Figure 6.15: Signature File Structure

Table 6.3: Symbols and Definitions for Signature File

symbol	definition
h	height of document tree
k	degree of document tree
m	average number of keywords in a leaf node
n_{doc}	number of documents in database
n_{false_drops}	number of false drops
n_{frame}	number of signature frames
p_{false_drop}	false drop probability
s_{block}	block size in bytes
s_{doc_id}	document identifier size in bytes
s_{elem_id}	element identifier size in bytes
s_{frame}	signature size of frame in bytes
t_{random}	random disk block access time in milliseconds
t_{seq}	sequential disk block access time in milliseconds
u	rate of unique index terms in children nodes
N_{drops}	number of actual drops

6.5.2.1 Storage Requirement

The storage requirement of the signature file structure is the sum of those required for the signatures and UID file.

$$S_{signature\_file} = S_{signature} + S_{uid\_file}. \tag{6.17}$$

The space needed for the signatures is

$$S_{signature} = n_{frame} * S_{frame}, \tag{6.18}$$

because the signature file consists of n_{frame} frames. The space required for one signature frame is

$$S_{frame} = \left\lceil \frac{s_{frame} * n_{doc} * N_{elem}}{s_{block}} \right\rceil, \tag{6.19}$$

where N_{elem} is the average number of signatures in a document. The storage requirement for the UID file is

$$S_{uid\_file} = \left\lceil \frac{(s_{doc\_id} + s_{elem\_id}) * n_{doc} * N_{elem}}{s_{block}} \right\rceil. \tag{6.20}$$

Now, we calculate N_{elem} for the SFAL and SFLN. Here, we assume that the document tree is full k-ary.

SFAL:

For the SFAL, we have to maintain as many signature files as the height of the document tree. The total number of signatures per document for the signature file of level i is the same as the number of nodes at level i. Thus,

$$N_{elem} = k^{i-1}. \tag{6.21}$$

SFLN:

In the SFAL, the number of signatures in a document is the same as the number of leaf nodes in a document tree. Thus,

$$N_{elem} = k^{h-1}. \tag{6.22}$$

6.5.2.2 Disk Access Time

The disk access time required to retrieve the UID list of a keyword is the sum of the signature frame access time and UID file access time.

$$T_{search} = T_{frame} + T_{uid\_file}. \tag{6.23}$$

The disk access time required to access a signature frame is

$$T_{frame} = t_{random} + (S_{frame} - 1) * t_{seq}. \tag{6.24}$$

The time needed to access satisfying UID's for the given keyword signature is

$$T_{uid\_file} = N_{drops} * t_{random} + p_{false\_drop} * n_{doc} * N_{elem} * t_{random}. \tag{6.25}$$

We assume that each UID requires one random disk access. Here, p_{false_drop} is false drop probability [58] and is defined as follows:

$$p_{false\_drop} = \frac{n_{false\_drops}}{n_{doc} * N_{elem} - N_{drops}}. \quad (6.26)$$

Now we calculate N_{drops} for the SFAL and SFLN. For N_{drops}, we can use the value $N_{post\_per\_key}$ obtained from the inverted index analysis. Thus, the number of actual drops for a given query looking for a keyword at level i is

$$N_{drops} = N_{post\_per\_key}$$

$$= \left\lceil \frac{N_{post\_per\_level} * n_{doc}}{n_{key}} \right\rceil. \quad (6.27)$$

SFAL:

The total number of index terms at level i in a document tree is

$$N_{post\_per\_level} = k^{h-1} * m * u^{h-i}. \quad (6.28)$$

SFLN:

The total number of index terms of the leaf nodes in a document tree is

$$N_{post\_per\_level} = k^{h-1} * m. \quad (6.29)$$

From this analysis, we can know that the ratio of the value of N_{drops} of SFAL to the value of that of the SFLN is

$$u^{h-i}.$$

6.5.2.3 Analytical Results

We compare the storage requirements of the SFAL and SFLN signature file schemes by using the parameters as shown in Table 6.4.

Table 6.4: Parameter Settings for Signature File

symbol	value
h	5
k	5
n_{doc}	varied
n_{frame}	32
p_{false_drop}	0.0001
s_{doc_id}	2
s_{elem_id}	2
s_{block}	1024
s_{frame}	1
t_{random}	19.06
t_{seq}	0.67
u	0.6

Figure 6.16 presents the storage requirements of the SFAL and SFLN for the various number of documents. It shows that the SFLN has less storage requirement than the SFAL.

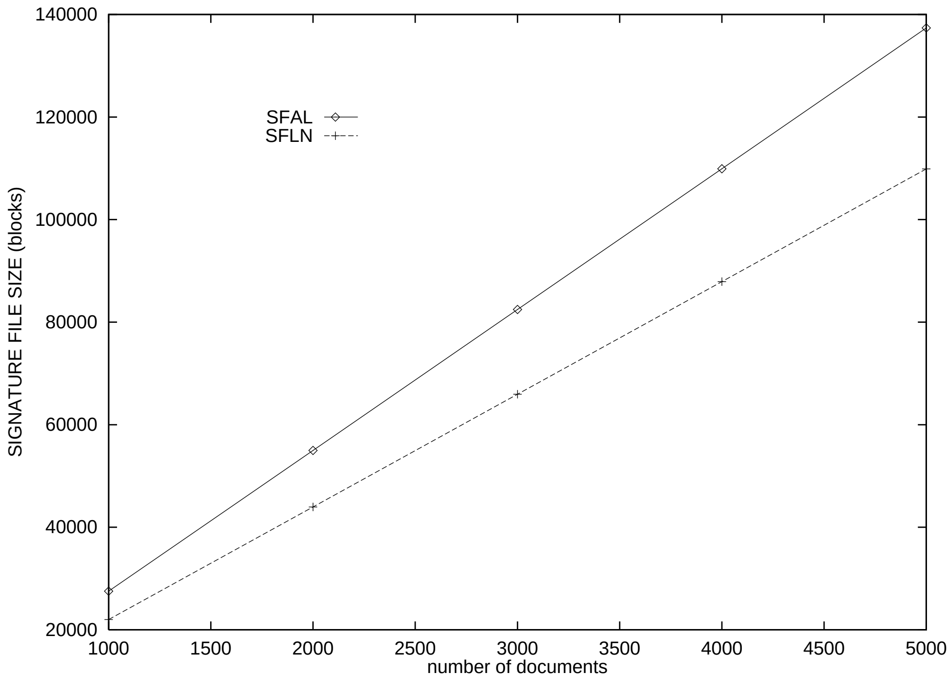

Figure 6.16: Signature File Storage Requirement

Figure 6.17 illustrates the average disk access times of the SFAL and SFLN, when u is 0.6, n_{doc} is 1000, and any arbitrary keyword in the document tree is accessed. It shows that the SFAL has slightly better performance in the average disk access time.

6.5.2.4 Experimental Results

Experiments have been performed on a SUN SPARC station with a local Conner CP30200 disk. We have assumed the database contains 1000 documents.

Figure 6.18 illustrates the results of the experiment obtained by accessing the signature file for a given keyword 100 times each and averaging the access times when u is 0.6. The results are quite similar to the analytical results.

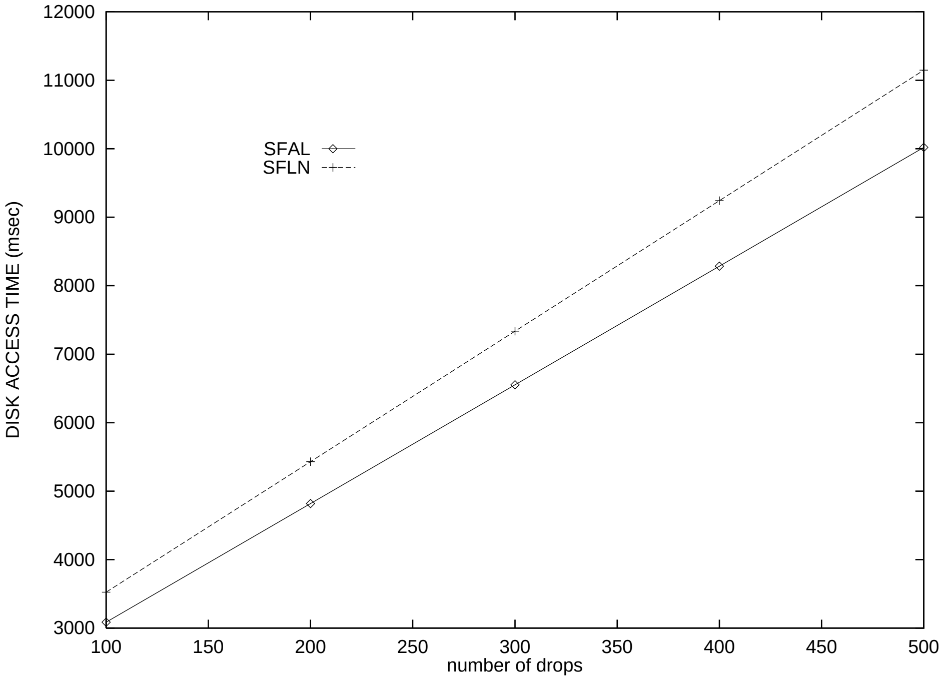

Figure 6.17: Average Disk Access Time

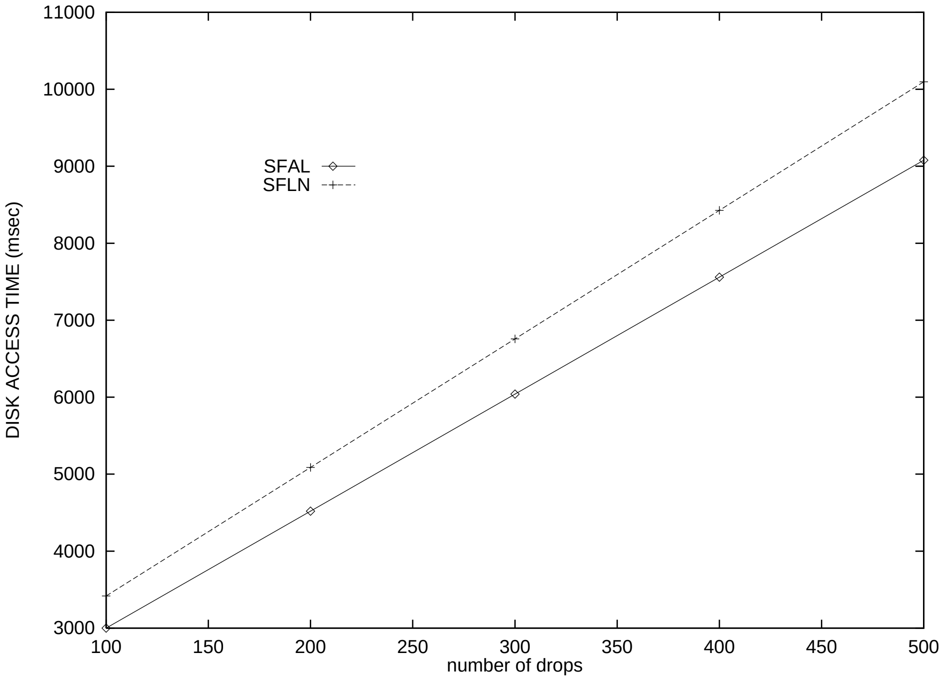

Figure 6.18: Experimental Average Disk Access Time

6.6 Summary

In this chapter, we have presented index structures which facilitate structure query processing for structured documents. We have proposed various kinds of new inverted index structures, and have analyzed index space requirements and disk access times of the schemes. Among the inverted index structures, the ANOR based on our UID scheme shows the best performance in the disk storage requirement. It also has the best performance in the disk access time when the elements of a document have some common keywords. This means that the ANOR can be the best choice because the difference of the storage requirements between the ANOR and others is large and it has a good disk access time. We also have proposed various signature file schemes for structured documents. Among the schemes, the SFLN based on our UID scheme has the best performance in the storage requirement while the SFAL shows the least disk access time. Since the difference of the disk access times between the SFLN and the SFAL is not large, the SFLN can be used when the storage requirement is critical. Experimentation also has shown the same results as with the analytical results for both the inverted index and signature file schemes.

Generally, the inverted index requires much more space than the signature file while the latter requires much more response time than the former. Thus, the inverted index is preferable when the response time is critical. The signature file can be used when the storage is not enough for the inverted index.

In order to process link expressions efficiently, we have proposed an index structure based on the join index.

Chapter 7

Database Schema for Structured Hyperdocuments

7.1 Introduction

Hypermedia documents contain composite nodes as well as single element nodes. Storing a single element node is simpler than that of the composite node. The discussion of the database schema will be focussed on the document which consists of multiple elements.

There have been two approaches in storing documents in databases. One is to store the whole document as a text string. The other is to partition a document into component elements and store them separately. In this chapter, we design database schemas for both approaches that support our identifier coding scheme. In order to compare our scheme with other schemes, we also design database schemas for the parse tree scheme [17] and the element locator scheme [6].

7.2 Partitioned Document Approach

We present a database schema for partitioned documents. Figure 7.1 is the DTD for a sample article and Figure 7.2 is a document that conforms to the DTD. We will use these for the explanation of the database schema.

```
<!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 7.1: Sample DTD for Article

The database schema proposed by Choy [24] has included the parent identifier in a database tuple corresponding to an element to represent the structure information as discussed in Chapter 2. The uniqueness of our database schema is that we do not need to store the document structure information such as the parent identifier or children identifiers in a database relation because we can obtain the structure information from the UID itself.

7.2.1 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 7.2, we need the relations as shown in Table 7.1,

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

Figure 7.2: Sample SGML Marked Document

Table 7.1: Relation TITLE

document id	element id	data
1	2	Structured Document Query Language
2	2	Index Structures for Structured Documents

Table 7.2: Relation NAME

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

Table 7.2, Table 7.3, Table 7.4, and Table 7.5. Each relation has three attributes (*document id*, *element id*, *data*). The element identifiers (*element id*) of the relations have been assigned assuming that the document tree is 5-ary. Even though we assign the element identifiers using our UID scheme, the PTS can use other identifier coding methods.

A sample parse tree of the example is shown in Figure 7.3 using a binary tree. Each node has three elements: element type, child pointer, and sibling pointer. In order to process simple structure queries, we need an element hierarchy tree for each DTD to represent the relationship between elements as shown in Figure 7.4. The plus (+) attached to the element name, for example, section+, means that the

Table 7.3: Relation ABSTRACT

document id	element id	data
1	4	We describe a query language ...
2	4	We propose index structures ...

Table 7.4: Relation SECTION.TITLE

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

Table 7.5: Relation PARAGRAPH

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

element will appear in a document one or more times. For database access, we need an element mapping table consisting of the triple (*element type*, *element name*, *relation name*) as shown in Table 7.6. The element type, defined as a positive integer, is used in the parse tree.

7.2.2 Element Locator Scheme (ELS)

For this scheme, we can use the relations defined for the PTS. Even though the element identifiers of the relations have been assigned by using our UID scheme, 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. Even though we assign the element identifiers using our UID scheme, the ELS can use other schemes. This scheme also needs the element hierarchy tree as shown in Figure 7.4.

Table 7.6: Element Mapping Table

element type	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

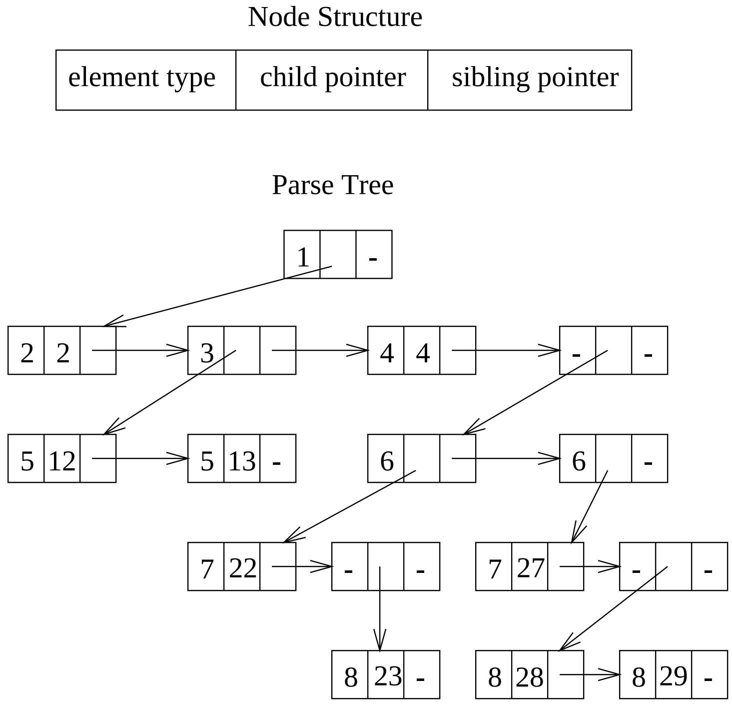

Figure 7.3: Sample Parse Tree

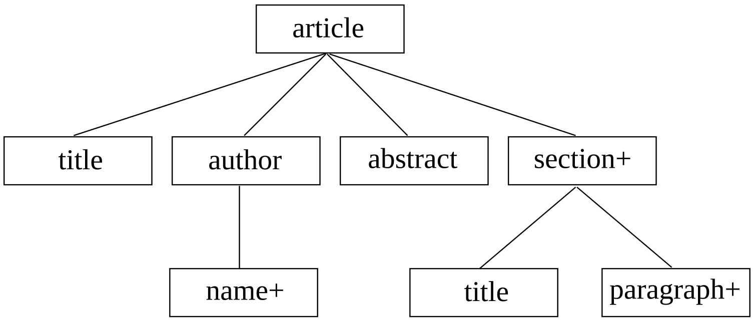

Figure 7.4: Element Hierarchy Tree

7.2.3 Identifier Coding Scheme (ICS)

In the ICS, we use the same relation tables 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. The number_of_children_table for the sample document is presented in Table 7.7. It is used during the evaluation of the forward branch expression in order to know the UID's of the descendent nodes of a node. We also need the element hierarchy tree obtained from the DTD as shown in Figure 7.4.

Table 7.7: Number of Children Table

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

The evaluation of a path expression, i.e., branch, link, and list expression, is to find the UID's of nodes reachable through branches or links from a given set of nodes. By adopting the ICS, most of the structure queries are performed using the *parent* and *child* function.

7.3 Whole Document Approach

For this approach, we can use the ICS with some changes. For the document data, we need only one relation with two attributes, (*document identifier*, *document data*). The *document data* will contains the whole document as a text string. We also need an element hierarchy tree as shown in Figure 7.4 for each DTD to represent the relationship between elements. We are also required to have the information about where each element starts within a document and the length in bytes to process structure queries. This relation consists of four attributes, (*document identifier*, *element identifier*, *start offset*, *length*). In order to represent the number of children of each internal node, the number_of_children_table should be maintained as presented in Table 7.7.

7.4 Storing Single Element Node

Hypermedia documents contain single element nodes which consist of a single element. For these nodes, we need not assign element identifiers, and they are stored into the database as the triple (*data type*, *document identifier*, *node data*). The *data type* represents the type of the node such as text, video, image, audio, or animation. When the data type of the node is multimedia such as image or video, we can store them using the multimedia data type supported by the database system.

7.5 Summary

In this chapter, we have designed database schemas supporting our scheme for both the partitioned document approach and whole document approach. In order to compare the query processing cost of our scheme with other schemes, we have also designed database schemas for the parse tree scheme and element locator scheme. Our database schemas do not store the document structure information such as the parent identifier or children identifiers in a database tuple because we can know the structure information from the UID itself. The database schemas will be used in the next chapter to compare the storage requirements and disk access times of the three schemes.

Chapter 8

Structure Query Processing

8.1 Introduction

In Chapter 7, we have presented database schemas for the parse tree scheme (PTS), element locator scheme (ELS), and identifier coding scheme (ICS). In the ICS, the general query processing procedure for the complex structure queries combined with the content query is as follows:

1. Access the element hierarchy tree for each document type specified by the query.
2. Get the element types specified by the query using the element hierarchy tree to know which types of elements will be accessed from the inverted index.
3. Access the ANOR inverted index for the given keywords in the query.
4. For each keyword, get the UID list A of the document elements containing the keyword from the inverted list.

5. For each keyword, get the UID list B for the specified element types from the UID list A using the following function:

$$\text{INDEX}[0] \cup \text{INDEX}[+] \cup \text{INDEX}[-].$$

6. Get the UID list C by ORing or ANDing all the UID list B's obtained from Step 3 as specified by the query. Now the UID list C contains the UID's of the elements of the specified element types that contain the given keywords.

7. Evaluate the path expressions specified by the query to get the target UID list D of the document elements.

8. Return the target UID list D of the document elements.

We have assumed that the indices are maintained using the ANOR inverted indexing scheme. The query processing procedures for other types of queries are simpler than those for the complex structure queries, and they can be defined similarly.

In this chapter, we analyze the storage requirements and the query processing costs of the path expression for the PTS, ELS, and ICS.

8.2 Path Expression Evaluation

The evaluation of a path expression is to locate the UID's of elements reachable through branches or links from a given set of UID's as specified in the expression. Brief procedures for the evaluation of the branch expression, link expression, and list expression in the ICS are given below.

```
Procedure: EvaluateBranchExpression
  Input: UID list
  Output: target UID list
  Case of Forward Branch Expression:
    1. Access the number_of_children_table.
    2. Calculate the target UID's using the child function
       and number_of_children_table.
    3. Return the target UID list.
  Case of Backward Branch Expression:
    1. Calculate the target UID's using the parent function
    2. Return the target UID list.


Procedure: EvaluateLinkExpression
  Input: UID list
  Output: target UID list
  1. Access the join index to obtain the target UID list.
  2. Return the target UID list.


Procedure: EvaluateListExpression
  Input: UID list
  Output: target UID list
  1. Calculate the target UID's using the child function.
  2. Return the target UID list.
```

In this section, we explain the evaluation of the path expression in the PTS, ELS, and ICS in detail.

8.2.1 Forward Branch Expression

For the evaluation of the forward branch expression in the PTS, we have to access parse trees of the related documents. Assume that we evaluate a forward branch expression d.branch[1]. To evaluate this expression, we need the parse trees of the documents instantiated to the document variable d. Thus, we have to retrieve as many parse trees as the number of documents of d.

Similarly, the LOC tables of the related documents should be accessed in the ELS. Thus, the number of LOC tables to be accessed is the same as in the PTS.

In the ICS, we have to access the table describing the number of children of each document. Thus, the number of children tables to be accessed is the same as in the PTS.

8.2.2 Backward Branch Expression

Like the forward branch expression evaluation, it is required to access parse trees in the PTS. For example, in order to evaluate d.branch[-1], we have to access as many parse trees as the set size of d.

In the ELS, the LOC tables of the documents specified by the branch expression should be retrieved. Thus, the number of LOC tables to be accessed is the same as in the PTS.

However, in the ICS, the UID's of the ancestors can be obtained using the *parent* function. That is, the UID's of the ancestors can be obtained directly from the UID's given by the document variable d.

8.2.3 Link Expression

In order to support hypertext navigation, we use the join index [73]. If the hypertext network is an acyclic graph, the evaluation of link expressions is not difficult even though complicated link expressions such as link[*] and link[+] require much time. However, if the hypertext network is a cyclic graph, we need a more complicated algorithm. We do not generate a UID more than once during the evaluation of the link expression to avoid an infinite loop.

8.2.4 List Expression

In the PTS, we have to access parse trees of the documents we are currently evaluating. For example, in order to evaluate d.section[1], we should access all the parse trees whose document elements are specified by the document variable d.

Similarly, in the ELS, we need the LOC tables of the documents which are instantiated to the document variable d.

However, in the ICS, we do not need to access the database because we can calculate the UID's of the children of the elements in d by the *child* function.

8.3 Cost Comparison

We analyze the space requirements and disk access times of the PTS, ELS, and ICS for the partitioned database schema. We consider only the cost of identifying required documents or elements because the cost of retrieving data from the database is the same for all schemes. The symbols in Table 8.1 are used in the space and time expression.

Table 8.1: Symbols and Definitions

symbol	definition
h	height of document tree
k	degree of document tree
s_{entry}	table entry size in bytes
s_{block}	block size in bytes
s_{set}	set size of document variable
t_{random}	random disk block access time in milliseconds
t_{seq}	sequential disk block access time in milliseconds

8.3.1 Storage Requirement

The PTS, ELS, and ICS use the same database relations for storing document elements. Besides these, the PTS maintains the parse tree, element_hierarchy_tree, and element_mapping_table. The ELS maintains the element_hierarchy_tree and LOC table. The ICS maintains the element_hierarchy_tree and number_of_children_table. The element_mapping_table and element_hierarchy_tree are maintained one for each DTD. Thus, we can ignore the storage cost for these tables.

The parse tree, LOC table, and number_of_children_table are maintained one for each document. Thus, the cost of storage depends on these tables. As shown in Table 7.7, the number_of_children_table has two entries for each internal node of the document tree. Here, we assume that the tree is full k-ary ($k > 1$) for easy analysis. We also assume that each entry of the table needs the same number of bytes. Then, the number of entries for the number_of_children_table is

$$2 * number\_of\_internal\_nodes.$$

The number of internal nodes of a k-ary tree of height h is

$$\sum_{i=0}^{h-2} k^i = \frac{k^{h-1} - 1}{k - 1}.$$

Thus, the number of disk blocks needed for a number_of_children_table in the ICS is

$$S_{ics} = \left\lceil \frac{2 * \frac{k^{h-1}-1}{k-1} * s_{entry}}{s_{block}} \right\rceil. \tag{8.1}$$

In the ELS, the number of elements of the LOC table is

$$2 * number\_of\_nodes.$$

Here, we also assume that the entries have the same bytes even though the LOC may require more space than the identifier. Thus, the number of disk blocks for the LOC table in the ELS is

$$S_{els} = \left\lceil \frac{2 * \frac{k^{h}-1}{k-1} * s_{entry}}{s_{block}} \right\rceil. \tag{8.2}$$

In the PTS, the parse tree maintains a node for each internal and leaf node. Each internal node has to contain at least $(k + 1)$ entries, one for element type and k for pointers to children. The leaf node also needs at least one entry to represent a UID of the element. Thus the number of entries for the PTS is

$$number\_of\_leaf\_nodes + (k + 1) * number\_of\_internal\_nodes.$$

Then, the number of entries for a parse tree is

$$k^{h-1} + (k+1) * \sum_{i=0}^{h-2} k^i$$

$$= k^{h-1} + \frac{(k+1)(k^{h-1}-1)}{k-1}.$$

The number of disk blocks required for a parse tree in the PTS is

$$S_{pts} = \left\lceil \frac{\left(k^{h-1} + \frac{(k+1)(k^{h-1}-1)}{k-1}\right) * s_{entry}}{s_{block}} \right\rceil . \tag{8.3}$$

8.3.2 Disk Access Time

8.3.2.1 Forward Branch Expression

In the PTS, we have to access parse trees as many 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. Thus, the disk access time is

$$T_{pts} = s_{set} * t_{random} + s_{set} * (S_{pts} - 1) * t_{seq}. \tag{8.4}$$

We assume that the parse tree of a document is stored consecutively in disk so that it requires only one disk seek.

In the ELS, the LOC table should be accessed as many as the set size of d. Thus, the disk access time is

$$T_{els} = s_{set} * t_{random} + s_{set} * (S_{els} - 1) * t_{seq}. \tag{8.5}$$

In the ICS, the number_of_children_table should be accessed as many as the set size of d. Thus, the disk access time is

$$T_{ics} = s_{set} * t_{random} + s_{set} * (S_{ics} - 1) * t_{seq}. \tag{8.6}$$

8.3.2.2 Backward Branch Expression

In the PTS, the number of parse trees which should be accessed to evaluate a backward branch expression such as d.branch[-1] is the same as the set size of d. Thus, the time required to access disk is

$$T_{pts} = s_{set} * t_{random} + s_{set} * (S_{pts} - 1) * t_{seq}. \tag{8.7}$$

In the ELS, the disk access time is the same as the evaluation of the forward branch expression.

$$T_{els} = s_{set} * t_{random} + s_{set} * (S_{els} - 1) * t_{seq}. \tag{8.8}$$

However, in the ICS, we do not need to access disk because we can get the UID's of the ancestors by the *parent* function. Therefore,

$$T_{ics} = 0. \tag{8.9}$$

8.3.2.3 Link Expression

In order to evaluate link expressions, join indices [73] should be maintained for all the PTS, ELS, and ICS. Thus, the time required to evaluate the link expression is the same for all schemes.

8.3.2.4 List Expression

In order to evaluate a list expression in the PTS, parse trees should be accessed. For example, if we evaluate d.section[3], we have to access parse trees as many as the set size of d. Thus, the time required to access disk is

$$T_{pts} = s_{set} * t_{random} + s_{set} * (S_{pts} - 1) * t_{seq}. \tag{8.10}$$

In the ELS, the time needed to access disk is

$$T_{els} = s_{set} * t_{random} + s_{set} * (S_{els} - 1) * t_{seq}. \tag{8.11}$$

However, in the ICS, we do not need to access disk, and therefore,

$$T_{ics} = 0. \tag{8.12}$$

8.3.3 Analytical Results

We compare the space and time requirements by setting the parameters in the above expressions. The parameter settings are shown in Table 8.2.

Figure 8.1 illustrates the storage requirements of the PTS, ELS, and ICS, when k is 5. The ICS shows the best storage utilization while the PTS and ELS show similar results (same curve in the figure). It shows that the difference increases as the height of the tree increases.

Figure 8.2 presents the disk access times of the PTS, ELS, and ICS to evaluate the branch expression (forward branch and backward branch) or link expression,

Table 8.2: Parameter Settings

symbol	value
h	5
k	5
s_{entry}	4
s_{block}	1024
s_{set}	10
t_{random}	19.06
t_{seq}	0.67

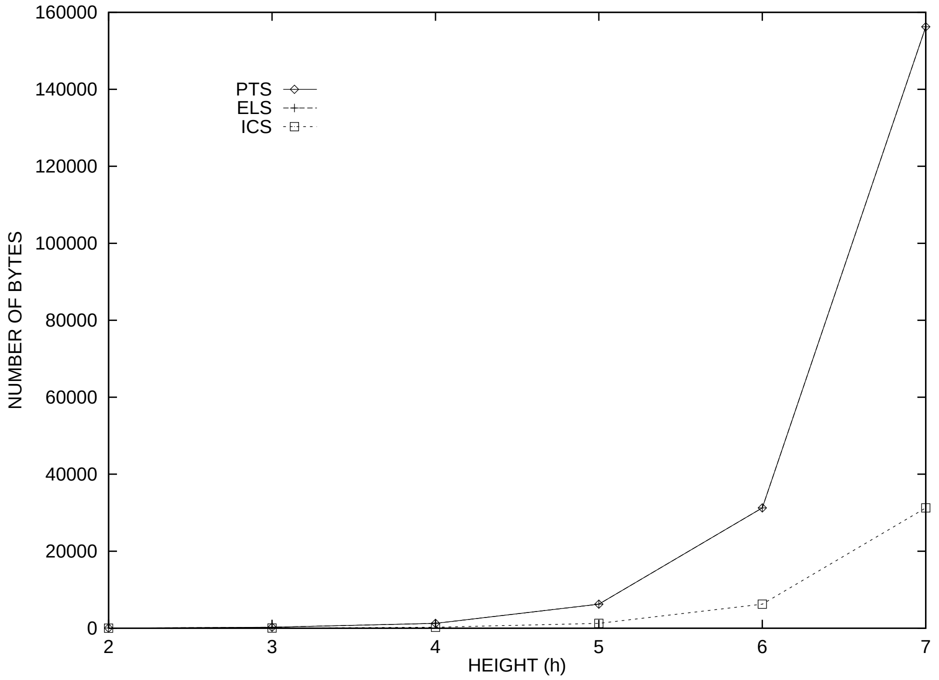

Figure 8.1: Storage Requirement

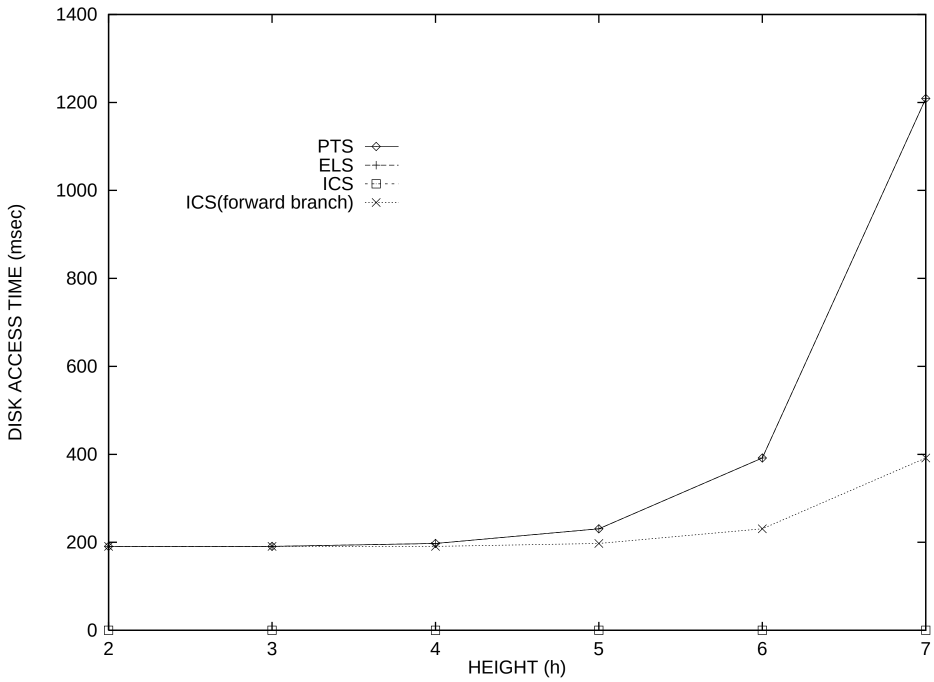

Figure 8.2: Disk Access Time

when k is 5 and the set size (number of documents to be processed) is 10. The ICS has two curves in the figure for the forward branch expression and other expressions respectively. The backward branch expression and the list expression in the ICS do not require disk access and marked as 0. As shown in the figure, the ICS has the best disk access time while the PTS and ELS show similar results (same curve in the figure). It also shows that the difference increases as the height of the tree increases.

8.3.4 Experimental Results

Experiments have been performed on a SUN SPARC station with a local Conner CP30200 disk. Figure 8.3 illustrates the results of the experiment obtained by accessing the tables 100 times each and averaging the access times when k is 5 and

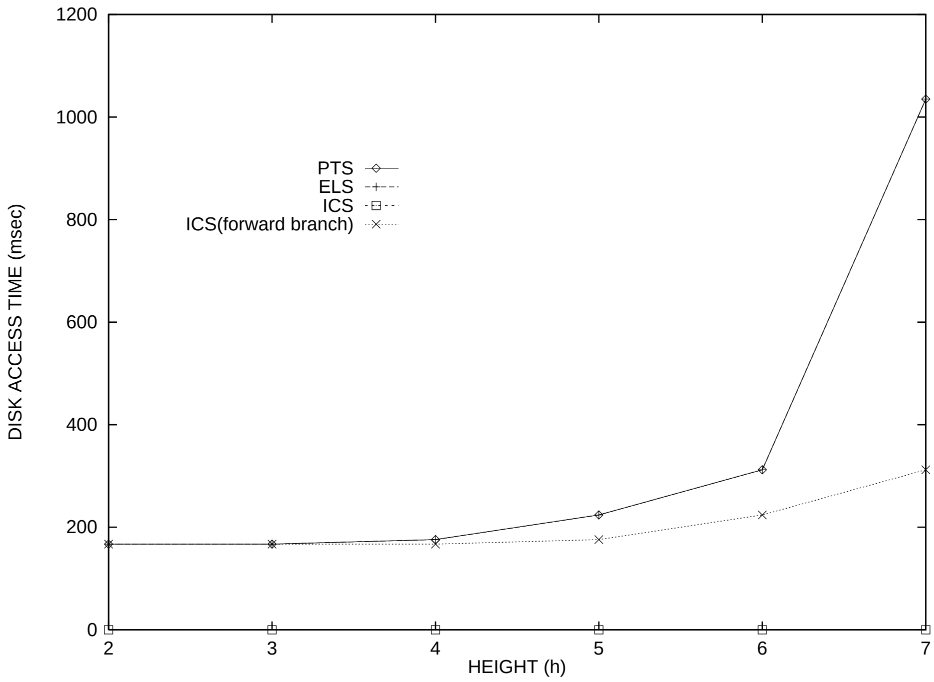

Figure 8.3: Experimental Disk Access Time

the set size is 10. The results are quite similar to the analytical results. The ICS shows the best disk access time for both of the forward branch expression and other expressions (backward branch and list) while the PTS and ELS show similar results (same curve in the figure).

8.4 Summary

In this chapter, we have analyzed the storage requirements and disk access times of the PTS, ELS, and ICS for structure query processing and have presented space and time expressions. Using the space and time expressions, we have compared the costs of the storage requirement and disk access time, and have shown that our scheme has the best performance among them. Experimentation also has shown similar results. A summary of the comparison is presented in Table 8.3.

Table 8.3: Performance Comparison

storage/disk access time	comparison
storage requirement	ICS is the best
forward branch expression evaluation	ICS is the best
backward branch expression evaluation	ICS is the best
link expression evaluation	same
list expression evaluation	ICS is the best

Chapter 9

Structured Video Documents

9.1 Introduction

In the previous chapters, we have proposed a structure query language, index structures, and a query processing scheme. Like text documents, video documents have a hierarchical logical structure of video elements such as segments, events, and shots [62] [63] [70]. In this chapter, we discuss how to apply our schemes to structured video documents.

9.2 Structured Video Document Model

Video documents have a hierarchical logical structure like text documents. The structure of video documents has been studied in [62] [63] [70]. The video elements, such as segments, events, and shots, compose a video which has a hierarchical structure as shown in Figure 9.1.

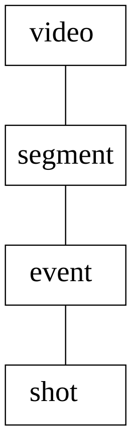

Figure 9.1: Video Element Hierarchy

The video document model, represented as a tree, forms a conceptual video structure on which a query language is based. An example of the video document tree according to the video element hierarchy of Figure 9.1 is illustrated in Figure 9.2.

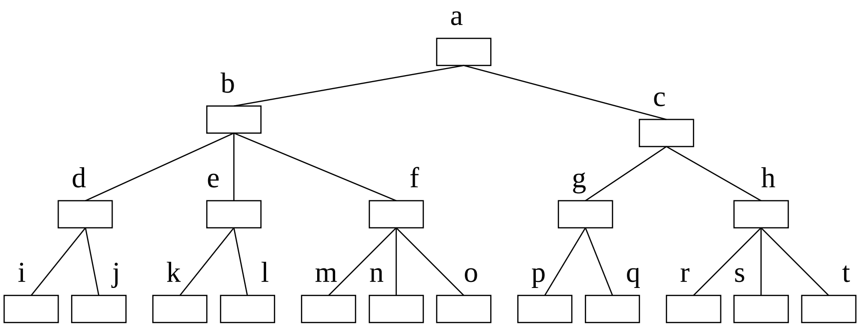

Figure 9.2: Example Video Document Tree

9.3 Video Structure Queries

We can generate structure queries for video documents using the path expression of the SDQL defined in Chapter 4. The following show some example queries.

- Find the events which have a shot about "war."

```
find e
from v:video, e:v.segment.event
where exists(e.branch[1] contains "war")
```

- Find the segments whose last event contains "farewell."

```
find e.branch[-1]
from v:video, e:v.segment.event[>]
where e contains "farewell"
```

- Find the first shot of the events which contain "mountain."

```
find e.shot[1]
from v:video, e:v.segment.event
where e contains "mountain"
```

- Find the video segments whose first shot contains "fish" and last event contains "animal."

```
find s
from v:video, s:v.segment, e:s.event[>], ss:s.event.shot[1]
where (ss contains "fish") and (e contains "animal")
```

- Find the first shot of the videos with attribute creation-date = "1995."

```
find v.shot[1]
from v:video
where v.creation-date = "1995"
```

We can also develop structure queries on the video documents belonging to the hyperdocument graph.

- Find all nodes that have a hypertext link to this_video.

```
find n
from n:node, v:this_video
where v in n.link[1]
```

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

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

- Find the last shot of the video that is linked by an article with last section containing the keyword "mountain."

```
find s
from v:video, s:v.shot[>], a:article, ss:a.section[>]
where (v in a.branch[*].link[1]) and (ss contains "mountain")
```

- Find the title of the documents created by the same authors as this_video.

```
find t
from d:*, t:d.title, v:this_video
where v.author in d.author
```

- Find all multimedia nodes which are referenced by the nodes that have a hypertext link to this_video.

```
find m
from m:multimedia, n:node, v:this_video
where (m in n.link[1]) and (v in n.link[1])
```

9.4 Video Element Identifier

We interpret a video structure as a k-ary tree [32] and assign each element a UID according to the level-order traversal order. For example, for the video document tree of Figure 9.2, we assign UID's as Table 9.1 assuming a 3-ary tree. Also we can use the tree locator scheme instead of the tree order scheme.

9.5 Supporting Dynamic Video Documents

Video documents can also change over time like text documents. By using the Extended Structured Hyperdocument Model (ESHM) and the Two Level UID scheme defined in Chapter 5, we can support video document sharing and video document changes.

Table 9.1: Video Element Identifiers

element	UID	element	UID
a	1	k	17
b	2	l	18
c	3	m	20
d	5	n	21
e	6	o	22
f	7	p	23
g	8	q	24
h	9	r	26
i	14	s	27
j	15	t	28

9.6 Indexing Video Annotations

Text data can be associated with a video element by user annotations or captions. Users can annotate any video element at any level with text data, and keywords can be extracted from these text data. Keywords can also be obtained by other methods such as automated image analysis tools. Suppose that a video has four leaf nodes with keywords extracted from annotations as shown in Figure 9.3. Here, internal nodes also have annotated keywords.

From these annotations, we can build a video index structure. Since users want to access any kind of video element in the video tree, it is necessary to build an index structure supporting element access. 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:

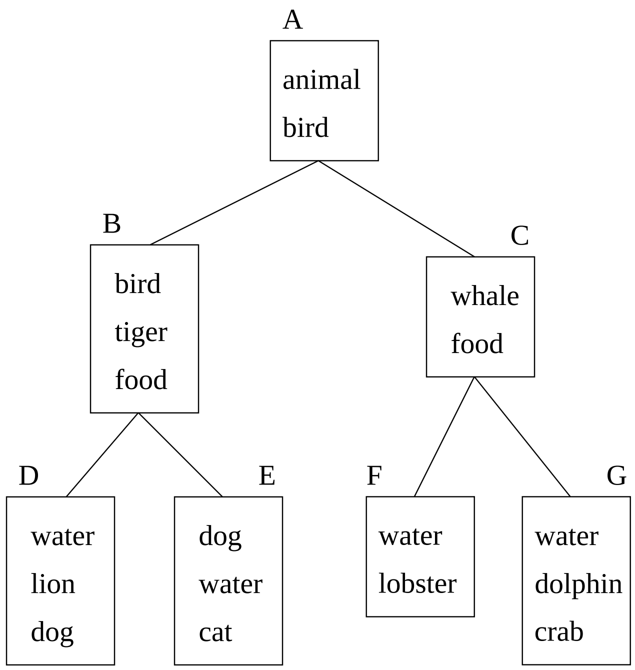

Figure 9.3: Video Document Tree with Index Terms

```
index(A) = {animal, bird, tiger, food, whale, water,
            lion, dog, cat, lobster, dolphin, crab}
index(B) = {bird, tiger, food, water, lion, dog, cat}
index(C) = {whale, food, water, lobster, dolphin, crab}
index(D) = {water, lion, dog}
index(E) = {dog, water, cat}
index(F) = {water, lobster}
index(G) = {water, dolphin, crab}
```

By using the fact that the child nodes of a node can have some keywords in common, we can use the ANOR (Inverted Index for All Nodes without Replication) scheme defined in Chapter 6. Figure 9.4 presents the video index structure constructed by using the ANOR scheme.

Thus, we can construct the inverted list for Figure 9.4 as follows:

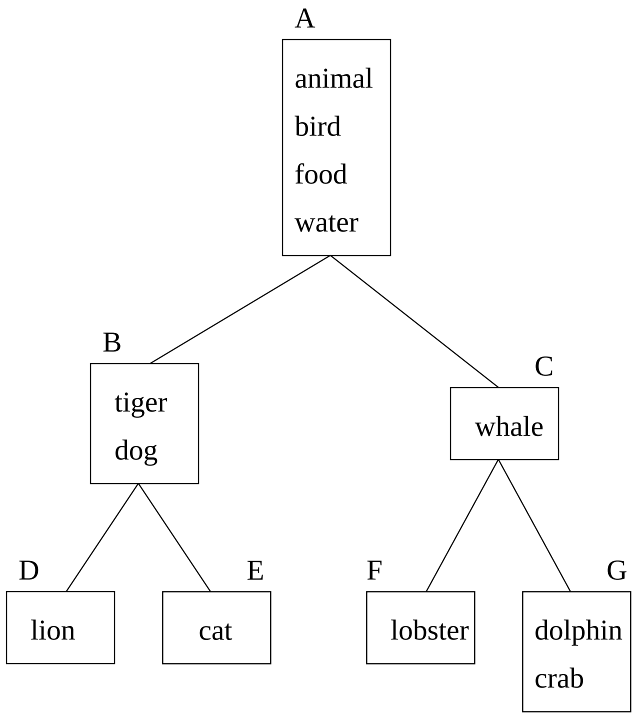

Figure 9.4: The ANOR Video Index

```
inverted-list(animal) = {A}
inverted-list(bird) = {A}
inverted-list(food) = {A}
inverted-list(water) = {A}
inverted-list(tiger) = {B}
inverted-list(dog) = {B}
inverted-list(whale) = {C}
inverted-list(lion) = {D}
inverted-list(cat) = {E}
inverted-list(lobster) = {F}
inverted-list(dolphin) = {G}
inverted-list(crab) = {G}
```

Even though we do not maintain the UID's of all the nodes which contain the keyword in the inverted list, we can calculate them by using the *parent* and *child* function defined in Chapter 5.

9.7 Video Database Schema

First, we need a relation to store video attributes such as title, author, security, rate, etc. In order to store video shots, we need a relation which has two attributes (*element identifier*, *shot data*). We can store some larger elements such as segments or events directly when we do not need to maintain indices on the subelements of those elements. A number_of_children_table is maintained for each video to represent the number of children of the internal node. The number_of_children_table is used during the forward branch expression processing. For the evaluation of the backward branch expression and list expression, we do not need the number_of_children_table.

9.8 Video Structure Query Processing

Video structure queries are processed using the inverted list and the UID's of video elements. We can use the same query processing scheme presented in Chapter 8.

9.9 Summary

In this chapter, we have discussed how to apply our query language, index structure, database schema, and query processing scheme to the management of structured video documents. We can use the path expression to make video structure queries. Using the UID's of video elements, structure queries can be performed efficiently like text documents. We have used the ANOR inverted indexing scheme for the video annotation index.

Chapter 10

System Architecture

In this chapter, we briefly describe the design of a structured hyperdocument management system which manages SGML documents using the Oracle database management system [64]. The system has been prototyped as a joint project between the Electronics and Telecommunications Research Institute of Korea and Syracuse University [11] [12] [13]. Figure 10.1 illustrates the global architecture of the system that consists of three main modules: the DTD processor, document processor, and query processor.

The DTD processor is the module which processes the DTD's conforming to the SGML syntax and consists of three submodules: the DTD parser, database schema generator, and database mapper generator as shown in Figure 10.2.

The DTD parser parses a DTD defined by the SGML and generates a parse tree. For the DTD parser, we have used the Amsterdam SGML Parser [74] [75] and added a parse tree generation module. From the parse tree, the database schema generator generates the relational database schema for the given DTD in SQL statements. The database schema is the partitioned schema discussed in Chapter 7. The

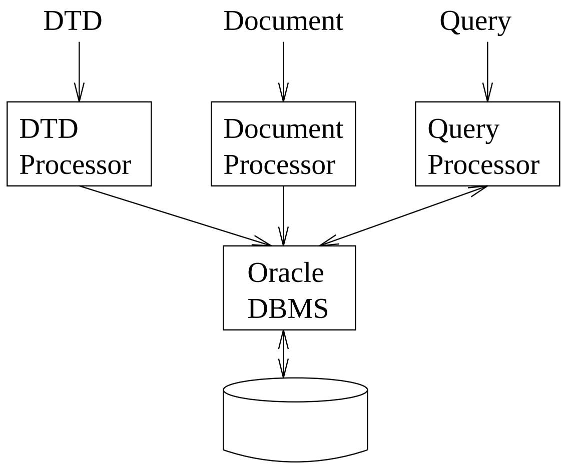

Figure 10.1: Structured Hyperdocument Management System

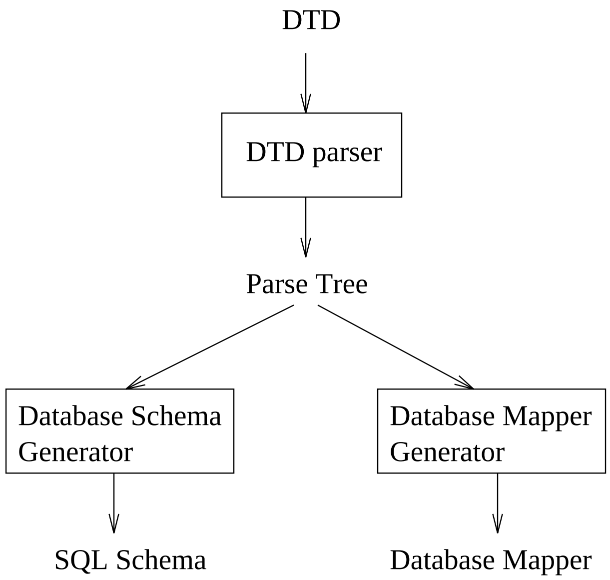

Figure 10.2: The DTD Processor

database mapper generator generates a database mapper which partitions a document instance into elements and stores them separately into the database relations.

The document processor processes a document instance conforming to a DTD and consists of three submodules: the document parser, element identifier generator, and index builder as shown in Figure 10.3.

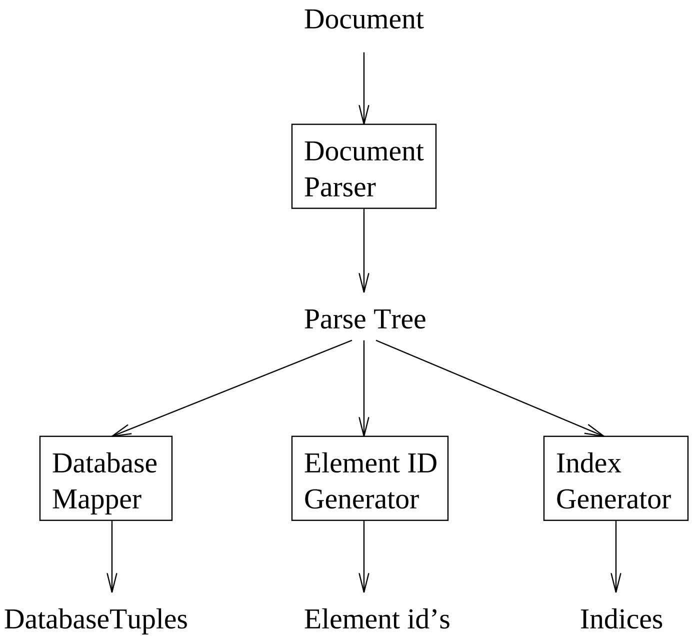

Figure 10.3: The Document Processor

The document parser generates a parse tree for a given document instance conforming the DTD. 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. The index builder extracts keywords from each element and inserts them into the ANOR inverted index structure. It also extracts hypertext links from the document and constructs the hypertext index using the join index.

The query processor evaluates the SHQL queries given by the user using the query processing scheme proposed in Chapter 8. It consists of three submodules: the lexical analyzer, query parser, and query evaluator. The lexical analyzer of the query processor has been developed using the Lex lexical analyzer generator [57], and the query parser has been implemented using the Yacc parser generator [57]. The query evaluator accesses the ANOR inverted index and processes path expressions.

Chapter 11

Conclusions and Future Research

11.1 Conclusions

Our Structured Hyperdocument Model (SHM) integrates the logical structure and hypertext structure of documents. By integrating the hierarchical model of documents with the hypertext model, we can build a structured hyperdocument management system with powerful query facilities. Based on the combined model, we have defined a new structure query language, called the Structured Hyperdocument Query Language (SHQL), which uses path expressions. The advantage of the SHQL is as follows: First, it is a user friendly language since it is declarative and has a similar format as the SQL. Second, because it has a powerful addressing capability, users can select any element or a group of elements within a document or a group of documents reachable from a node by using the path expression. Finally, it is a general hyperdocument query language supporting the content query, structure query, attribute query, and multimedia query. Thus, users have the advantage of navigating through hyperdocuments via logical paths and hypertext links and retrieving any interested portions of a document or documents by any kind of the queries or combined queries.

In order to support fast element access, we have proposed new indexing schemes for structured hyperdocuments based on the inverted index and signature file. For the fast hypertext navigation, we have used the join index. The number of the inverted list entries of the structured documents is exponential when the number of documents is large because every element of a document must be indexed. The index structures of this research have been designed to reduce the storage overhead considerably. We have developed the storage requirements and disk access times of the various indexing schemes. Among the inverted indexing schemes, the ANOR (inverted index without replication) has shown the best performance in the storage requirement. It also has the least disk access time when the child nodes of a node have some common keywords. The ANOR can be the best choice since the difference of the storage requirements between the ANOR and other schemes is great. Among the signature file schemes, the SFLN (signature file for leaf nodes only) has the least storage requirement while the SFAL (signature file for all levels) has the best disk access time. Since the difference of the disk access times between the SFLN and the SFAL is not large, the SFLN can be used when the storage requirement is critical.

To process structure queries efficiently, we have proposed a novel scheme which uses unique element identifiers (UID's). Our query processing scheme is especially efficient for the evaluation of the branch and list expression because they are processed by calculations without accessing the disk. We have presented relational database schemas for the identifier coding scheme (ICS), parse tree scheme (PTS), and element locator scheme (ELS). Using the database schema, we have analyzed and experimented the storage requirement and disk access time for each scheme, and have shown that the ICS has the best performance.

Our schemes can be applied to structured multimedia documents, such as video documents, as well as text documents since the data type of a node in the hyper-document graph is multimedia. In this research, we have used the path expression to represent video structure queries. In order to process video structure queries efficiently, we have used the element identifier coding scheme and the ANOR inverted indexing scheme.

11.2 Future Research

11.2.1 Extending Index Structures

The index structures in this research have been based on the boolean information retrieval model. That is, users generate queries on the document database using and-or combination of keywords. In the following paragraphs, we discuss how to expand our index structures to support ranked queries and the vector space model.

Ranked Queries:

Ranked queries can also be supported by using weighted keywords in some of the inverted indexing schemes such as ANWR (inverted index for all nodes with replication) and LNON (inverted index for leaf nodes only). However, in the ANOR (inverted index for all nodes without replication), it is more difficult to represent the weights of keywords for the individual document element because the keywords may be promoted to its ancestor nodes. Thus, we need a new weighting function which exploits this fact. That is, we can partition keywords of a document element into three classes, one representing ancestor node keywords, one representing its own keywords, and one representing descendent keywords. Then, we can define a different weighting function for each class.

Vector Space Model:

The vector space model [67] is one of the most frequently used information retrieval methods to support ranked queries. We have not considered the vector space model in this research. In order to support the vector space model the keyword weighting function and similarity calculation methods should be defined. In the case of the ANOR, the application of the vector space model may be more difficult than other schemes because of the reasons explained in the previous paragraph.

11.2.2 Supporting Document Versioning

Hypertext and database versioning is an important research issue and considerable research has been previously carried out [15] [44] [48] [53] [54]. However, there has been little research on the issue of structured document versioning. We can separate it into two different issues: DTD versioning and document instance versioning. The DTD versioning is similar to the database schema versioning, and the document instance versioning is similar to the database instance versioning.

DTD Versioning:

The DTD versioning is for allowing users to change a DTD which may have associated document instances in the database. The DTD versioning is more difficult since it may cause database schema changes.

Document Instance Versioning:

In the partitioned document approach, the document versioning is easier than that in the whole document approach. For the document instance versioning, we can adopt hypertext or database instance versioning techniques. The versioning operations that should be considered include version creation, selecting a specific version, and version promotion.

11.2.3 Managing Video Documents

The proposed scheme for structured video documents can be extended in the following two directions.

Extending the SHQL:

In this research, we have only considered the logical structure of video documents in the structure query language. Other structures such as synchronization relationships may exist in a video document. Thus, the structure query language can be extended to support structure queries based on the synchronization structure between different video elements or different media.

Video Composition Algebra:

Video composition algebra [76] [77] which is used to compose a video document from existing video elements is useful for video document creation. This algebra can be combined with the structure query language to build more powerful video document management system.

Appendix: The Structured Hyperdocument Query Language (SHQL) Definition

We use a variation of the Backus Naur Form (BNF) for the formal specification of the syntax of query statements. The following metasymbols are used.

```
< >  Non-terminal symbols are given in < >.
' '  Terminal symbols are given in ' '.
::=  This represents a production rule.
[ ]  The part placed between square brackets is optional.
{ }  All symbols placed between braces form a group.
...  This represents repetition.
|    This represents alternatives.
```

```
/* query definition */
<query> ::=
   find <target clause>
```

```
   from <range clause>
   [where <qualification clause>]

/* target clause definition */
<target clause> ::=
   <target expression> [{, <target expression>} ... ] | *
<target expression> ::=
   union (<target expression>, <target expression>
          [{, <target expression>} ... ]) |
   intersection (<target expression>, <target expression>
          [{, <target expression>} ... ]) |
   difference (<target expression>, <target expression>
          [{, <target expression>} ... ]) |
   <query expression>
<query expression> ::=
   <path head> [{. <path expression>} ... ] [. <path tail>]
<path head> ::=
   <path variable>
<path variable> ::=
   <identifier>
<path expression> ::=
   <path term> {<list term>}
<path term> ::=
   <branch term> | <link term> | <element term>
<branch term> ::=
   branch '[' <range> ']'
<range> ::=
    '*' | '+' | '/' | '-' | '<' | '>' | '0' |
```

```
   <integer> |
   <integer> - <integer> |
   <integer> - '>' |
<link term> ::=
   link '[' <range> ']'
<element term> ::=
   <element name>
<element name> ::=
   <identifier>
<list term> ::=
   <list> '[' <list range> [{, <list range>} ... ] ']'
<list> ::=
   <branch term> | <link term> | <element term>
<list range> ::=
   <positive integer> |
   <positive integer> '-' <positive integer> |
   '>'
<path tail> ::=
   element |
   attribute |
   content

/* range clause definition */
<range clause> ::=
   <range definition> [{, <range definition>} ... ] |
<range definition> ::=
   <path variable> ':' <range expression>
<range expression> ::=
```

```
   <query expression> |
   <document type> [{or <document type>} ...]
<document type> ::=
   <identifier>

/* qualification clause definition */
<qualification clause> ::=
   <predicate expression>
<predicate expression> ::=
   <simple predicate> <boolean operator> <predicate expression> |
   not <predicate expression> |
   <simple predicate>
<boolean operator> ::=
   and | or
<simple predicate> ::=
   <contain predicate> |
   <comparison predicate> |
   <exists predicate>
<contain predicate> ::=
   <query expression> contains <keyword expression>
<keyword expression> ::=
   <keyword> <boolean operator> <keyword expression> |
   <keyword>
<keyword> ::=
   '"' <identifier> '"'
<comparison predicate> ::=
   <query expression> <set comparator> <query expression> |
   <numeric expression> <numeric comparator> <numeric expression>
```

```
   <string expression> <string comparator> <string expression>
<set comparator> ::=
   in | not in
<numeric expression> ::=
   <order function> |
   <weight function> |
   <count function> |
   <real number>
<order function> ::= order ( <query expression> )
<weight function> ::= weight ( <query expression>, <keyword> )
<count function> ::= count ( <query expression> )
<numeric comparator> ::=
   = | < | > | <= | >= | !=
<string expression> ::=
   <type function> |
   <name function> |
   <string>
<type function> ::=
   type ( <query expression> )
<name function> ::=
   name ( <query expression> )
<exists predicate> ::=
   exists <query expression>
<string comparator> ::=
   equal | not_equal
```

Bibliography

[1] R. M. Akscyn, D. L. McCracken, and E. A. Yoder, "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations," *Communications of the ACM,* vol. 31, no. 7, pp. 820-835, July 1988.

[2] M. Anderegg, B. Elledge, J. Harford, D. Shackelford, and O. Toki, "User's Manual for Attributed Directed Graph Server," Department of Computer Science, University of North Carolina, May 1990.

[3] W. Appelt, *Document Architecture in Open Systems: The ODA Standard,* Springer Verlag, New York, 1991.

[4] W. Appelt and N. Tetteh-Lartey, "The Formal Specification of the ISO Open Document Architecture (ODA) Standard," *The Computer Journal,* vol. 36, no. 3, pp. 269-279, 1993.

[5] T. Arnold-Moore, M. Fuller, B. Lowe, J. Thom, and R. Wilkinson, "SGQL: A Data Model and Query Language for Structured Documents," CITRI TR 94-8, Collaborative Information Technology Research Institute, Melbourne, Australia, 1994.

[6] T. Arnold-Moore, M. Fuller, B. Lowe, J. Thom, and R. Wilkinson, "The ELF Data Model and SGQL Query Language for Structured Document Databases,"

CITRI TR 94-13, Collaborative Information Technology Research Institute, Melbourne, Australia, 1994.

[7] V. Balasubramanian, "State of the Art Review on Hypermedia Issues and Applications," Technical Report, Graduate School of Management, Rutgers University, 1993.

[8] T. Berners-Lee, R. Cailliau, J. Groff, and B. Pollermann, "World-Wide Web: The Information Universe," *Electronic Networking: Research, Applications and Policy,* vol. 2, no. 1, pp. 52-58, Meckler Publications, Westport, Connecticut, Spring 1992.

[9] T. Berners-Lee, R. Cailliau, N. Pellow, and A. Secret, "The World-Wide Web Initiative," Proceedings of the INET '93 Conference, 1993.

[10] 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.

[11] P. B. Berra, Y. K. Lee, and K. Yoon, "Multimedia Database Design for Heterogeneous Distributed Database Systems: 1st Interim Report," The New York State Center for Advanced Technology in Computer Applications and Software Engineering, Syracuse University, December 1994.

[12] P. B. Berra, Y. K. Lee, and K. Yoon, "Multimedia Database Design for Heterogeneous Distributed Database Systems: 2nd Interim Report," The New York State Center for Advanced Technology in Computer Applications and Software Engineering, Syracuse University, June 1995.

[13] P. B. Berra, Y. K. Lee, and K. Yoon, "Multimedia Database Design for Heterogeneous Distributed Database Systems: 3nd Interim Report," The New York State Center for Advanced Technology in Computer Applications and Software Engineering, Syracuse University, December 1995.

[14] E. Bertino, F. Rabitti, and S. Gibbs, "Query Processing in a Multimedia Document System," *ACM Transactions on Office Information Systems,* vol. 6, no. 1, pp. 1-41, January 1988.

[15] E. Bertino and L. Martino, *Object-Oriented Database Systems: Concepts and Architectures,* Addison-Wesley, 1993.

[16] G. E. Blake, T. Bray, and F. W. Tompa, "Shortening the OED: Experience with a Grammar-Defined Database," *ACM Transactions on Information Systems,* vol. 10, no. 3, pp. 213-232, July 1992.

[17] G. E. Blake, M. P. Consens, P. Kilpelainen, P. -A. Larson, T. Snider, and F. W. Tompa, "Text/Relational Database Management Systems: Harmonizing SQL and SGML," *Proceedings of the International Conference on Applications of Databases,* pp. 267-280, Vadstena, Sweden, June 1994.

[18] J. Boone, "A Survey of Data Models for Hypermedia," Technical Report TR 92-021, Department of Computer Science, University of North Carolina, April 1992.

[19] P. D. Bruza, "A Novel Aid for Searching in Hypermedia," *Proceedings of the First European Conference on Hypertext (ECHT '90),* pp. 109-122, Paris, France, November 1990.

[20] P. D. Bruza and T. P. van der Weide, "Stratified Hypermedia Structures for Information Disclosure," *The Computer Journal,* vol. 35, no. 3, pp. 208-219, 1992.

[21] J. F. Buford, "Evaluation of a Query Language for Structured Hypermedia Documents," *Proceedings of the Dartmouth Institute for Advanced Graduate Studies (DAGS) Conference on the Electronic Publishing and the Information Superhighway,* pp. 105-116, Boston, Massachusetts, May 1995.

[22] F. J. Burkowski, "An Algebra for Hierarchically Organized Text-Dominated Databases," *Information Processing and Management,* vol. 28, no. 3, pp. 333-348, 1992.

[23] B. Campbell and J. M. Goodman, "HAM: A General Purpose Hypertext Abstract Machine," *Communications of the ACM,* vol. 31, no. 7, pp. 856-861, July 1988.

[24] D. M. Choy, F. Barbic, R. H. Gueting, D. Ruland, and R. Zicari, "Document Management and Handling," *Proceedings of the IEEE '87 Office Automation Symposium,* pp. 241-246, 1987.

[25] S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, and A. Pathria, "Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and a System," *ACM Transactions on Office Information Systems,* vol. 4, no. 4, pp. 345-383, October 1986.

[26] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl, "From Structured Documents to Novel Query Facilities," *Proceedings of the 1994 ACM SIGMOD Conference,* pp. 313-324, Minneapolis, Minnesota, May 1994.

[27] V. Christophides and A. Rizk, "Querying Structured Documents with Hypertext Links using OODBMS," *Proceedings of the European Conference on Hypermedia Technology (ECHT '94),* pp. 186-197, Edinburgh, UK, September 1994.

[28] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski, "An Algebra for Structured Text Search and A Framework for its Implementation," Technical Report CS-94-30, Department of Computer Science, University of Waterloo, Waterloo, Canada, August 1994.

[29] C. Clifton, H. Garcia-Molina, and R. Hagmann, "The Design of a Document Database," Technical Report CS-TR-177-88, Department of Computer Science, Princeton University, September 1988.

[30] C. Clifton and H. Garcia-Molina, "Indexing a Hypertext Database," *Proceedings of the 16th International Conference on Very Large Data Bases,* pp. 36-49, Brisbane, Australia, 1990.

[31] Conner Periperials, Inc., "CP-30200 Specification Summary," 1995.

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

[33] C. Faloutsos, "Access Methods for Text," *ACM Computing Surveys,* vol. 17, no. 1, pp. 49-74, March 1985.

[34] C. Faloutsos, "Signature Files," In W. B. Frankes and R. Baeza-Yates, Ed., *Information Retrieval: Data Structures and Algorithms,* pp. 44-65, Prentice Hall, 1992.

[35] C. Faloutsos and H. V. Jagadish, "Hybrid Index Organizations for Text Databases," *Proceedings of the 3rd International Conference on Extending Database Technology,* Vienna, Austria, Mar. 1992, In *Computer Science Lecture Note 580: Advances in Database Technology - EDBT '92,* pp. 310-327, Springer-Verlag, 1992.

[36] W. B. Frankes and R. Baeza-Yates, Ed., *Information Retrieval: Data Structures and Algorithms,* Prentice Hall, 1992.

[37] R. Furuta and P. D. Stotts, "The Trellis Hypertext Reference Model," *Proceedings of the NIST Hypertext Standardization Workshop,* pp. 83-93, Gaithersburg, Maryland, January 1990.

[38] P. K. Garg, "Abstract Mechanisms in Hypertext," *Communications of the ACM,* vol. 31, no. 7, pp. 862-870, July 1988.

[39] C. F. Goldfarb, *The SGML Handbook,* Oxford University Press, Oxford, UK, 1990.

[40] G. H. Gonnet and F. W. Tompa, "Mind Your Grammar: a New Approach to Modeling Text," *Proceedings of the 13th International Conference on Very Large Data Bases,* pp. 339-346, Brighton, England, September 1987.

[41] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures,* Addison-Wesley, 1991.

[42] R. H. Guting, R. Zicari, and D. M. Choy, "An Algebra for Structured Office Documents," *ACM Transactions on Office Information Systems,* vol. 7, no. 4, pp. 123-157, April 1989.

[43] M. Gyssens and J. Paredaens, "A Grammar-Based Approach towards Unifying Hierarchical Data Models," *Proceedings of the 1989 ACM SIGMOD Conference,* pp. 263-272, Portland, Oregon, 1989.

[44] A. Haake, "CoVer: A Contextual Version Server for Hypertext Applications," *Proceedings of the 5th ACM Conference on Hypertext (Hypertext '93),* pp. 43-52, Seattle, Washington, November 1993.

[45] F. G. Halasz and M. Schwartz, "The Dexter Hypertext Reference Model," *Proceedings of the NIST Hypertext Standardization Workshop,* pp. 95-133, Gaithersburg, Maryland, January 1990.

[46] F. G. Halasz and M. Schwartz, "The Dexter Hypertext Reference Model," *Communications of the ACM,* vol. 37, no. 2, pp. 29-39, February 1994.

[47] D. Harman, E. Fox, R. Baeza-Yates, and W. Lee, "Inverted Files," In W. B. Frankes and R. Baeza-Yates, Ed., *Information Retrieval: Data Structures and Algorithms,* pp. 28-43, Prentice Hall, 1992.

[48] D. L. Hicks, J. J. Leggett, and J. L. Schnase, "Version Control in Hypermedia: An Open Systems Perspective," Technical Report TAMU-HRL 93-001, Hypermedia Research Lab., Texas A&M University, January 1993.

[49] *ISO 8613. Information Processing - Text and Office Systems - Office Document Architecture (ODA) and Interchange Format,* International Organization for Standardization, 1993.

[50] *ISO 8879. Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML),* International Organization for Standardization, 1986.

[51] *ISO/IEC 10744. Information Technology - Hypermedia/Time-Based Structuring Language (HyTime),* International Organization for Standardization, 1992.

[52] M. Kifer, W. Kim, and Y. Sagiv, "Querying Object-Oriented Databases," *Proceedings of the 1992 ACM SIGMOD Conference,* pp. 393-402, San Diego, California, 1992.

[53] W. Kim and H. -T. Chou, "Versions of Schema for Object-Oriented Databases," *Proceedings of the 14th International Conference on Very Large Data Bases,* pp. 148-159, Los Angeles, California, August 1988.

[54] W. Kim, *Introduction to Object-Oriented Databases,* The MIT Press, 1990.

[55] W. E. Kimber, "HyTime and SGML: Understanding the HyTime HYQ Query Language," Available via anonymous ftp at ftp.ifi.uio.no /pub/SGML/HyTime, August 1993.

[56] D. B. Lange, "A Formal Model of Hypertext," *Proceedings of the NIST Hypertext Standardization Workshop,* pp. 145-166, Gaithersburg, Maryland, January 1990.

[57] J. R. Levine, T. Mason, and D. Brown, *Lex & Yacc,* O'Reilly & Associates, 1992.

[58] Z. Lin and C. Faloutsos, "Frame-Sliced Signature Files," *IEEE Transactions on Knowledge and Data Engineering,* vol. 4, no. 3, pp. 281-289, June 1992.

[59] I. A. Macleod, "Storage and Retrieval of Structured Documents," *Information Processing and Management,* vol. 26, no. 2, pp. 197-208, 1990.

[60] I. A. Macleod, "A Query Language for Retrieving Information from Hierarchic Text Structures," *The Computer Journal,* vol. 34, no. 3, pp. 254-264, 1991.

[61] S. R. Newcomb, N. A. Kipp, and V. T. Newcomb, "The HyTime: Hypermedia/Time-Based Document Structuring Language," *Communications of the ACM,* vol. 34, no. 11, pp. 67-83, November 1991.

[62] D.-Y. Oh, S. S. Kumar, and P. V. Rangan, "Content-Based Inter-Media Synchronization," *Proceedings of the SPIE Conference on Multimedia Computing and Networking,* vol. 2417, pp. 202-214, San Jose, California, February 1995.

[63] E. Oomoto and K. Tanaka, "OVID: Design and Implementation of a Video-Object Database System," *IEEE Transactions on Knowledge and Data Engineering,* vol. 5, no. 4, pp. 629-643, August 1993.

[64] Oracle Corporation, *ORACLE7 Reference Manual*, 1992.

[65] H. V. D. Parunak, "Ordering the Information Graph," In E. Berk and J Devlin, Ed., *Hypertext/Hypermedia Handbook,* pp. 299-325, McGraw-Hill, 1991.

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

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

[68] G. Salton, *Automatic Text Processing,* Addison-Wesley, 1989.

[69] H. A. Schutt and N. A. Streitz, "Hyperbase: A Hypermedia Engine Based on a Relational Database Management System," *Proceedings of the First European Conference on Hypertext (ECHT '90),* pp. 95-108, Paris, France, 1990.

[70] B. Simonnot, "A Cooperation Model for Video Document Retrieval," *Proceedings of the SPIE Conference on Multimedia Computing and Networking,* vol. 2417, pp. 307-317, San Jose, California, February 1995.

[71] K. Summers, "Toward a Taxonomy of Logical Document Structures," *Proceedings of the Dartmouth Institute for Advanced Graduate Studies (DAGS) Conference on the Electronic Publishing and the Information Superhighway,* pp. 124-133, Boston, Massachusetts, May 1995.

[72] F. Tompa, "A Data Model for Flexible Hypertext Database Systems," *ACM Transactions on Office Information Systems,* vol. 7, no. 1, pp. 85-100, January 1989.

[73] P. Valduriez, "Join Indices," *ACM Transactions on Database Systems,* vol. 12, no. 2, pp. 218-246, June 1987.

[74] J. Warmer and S. Van Egmond, "The Implementation of the Amsterdam SGML Parser," *Electronic Publishing,* vol. 2, no. 2, pp. 65-90, John Wiley & Sons, July 1989.

[75] J. Warmer and H. Van Vliet, "Processing SGML Documents," *Electronic Publishing,* vol. 4, no. 1, pp. 3-26, John Wiley & Sons, March 1991.

[76] R. Weiss, A. Duba, and D. K. Gifford, "Content-Based Access to Algebraic Video," *Proceedings of the IEEE International Conference on Multimedia Computing and Systems,* pp. 140-151, 1994.

[77] R. Weiss, A. Duba, and D. K. Gifford, "Composition and Search with a Video Algebra," *IEEE Multimedia,* pp. 12-25, Spring 1995.

[78] N. Yankelovich, B. Haan, N. Meyrowitz, and S. Drucker, "Intermedia: The Concept and the Construction of a Seamless Information Environment," *Computer,* pp. 81-96, January 1988.

[79] J. Zobel, J. A. Thom, and R. Sacks-Davis, "Efficiency of Nested Relational Document Database Systems," *Proceedings of the 17th International Conference on Very Large Data Bases,* pp. 91-102, Barcelona, Spain, September 1991.

VITA

NAME OF AUTHOR: Yong Kyu Lee

PLACE OF BIRTH: Choong-Nam, Korea

DATE OF BIRTH: September 5, 1958

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea

Dongguk University, Seoul, Korea

DEGREES AWARDED:

Master of Science in Computer Science, 1988, KAIST

Bachelor of Engineering in Computer Science, 1986, Dongguk University

AWARDS AND HONORS:

State Scholarship, Korea, 1986-1988

Jung-Su Scholarship, Korea, 1984-1986

University Scholarship, Dongguk University, 1983-1984

PROFESSIONAL EXPERIENCE:

Research Assistant, The New York State Center for Advanced Technology in Computer Applications and Software Engineering (CASE), 1994-1996

Senior Researcher/Project Leader/Section Head, Institute for Defense Information Systems (IDIS), Seoul, Korea, 1988-1993

Research Assistant, Computer Science Department, KAIST, 1986-1987