The Use of a Combined Text/Relational Database System to Support Document Management

CS-96-07

January 1996

Kar Yan Ng *

Department of Computer Science

University of Waterloo

Waterloo, Ontario, Canada

kyng@watsol.uwaterloo.ca

*This is the thesis which embodies the results of my research done in partial fulfilment of the requirements for the degree of Master of Mathematics in Computer Science at the Department of Computer Science, University of Waterloo

Abstract

In this thesis, we study the problem of representing and manipulating a document to facilitate browsing, editing, string/content searches and document assembly.

Two major data models in which documents are represented and stored are :

1. a relational *data model, where all text contents in a document are represented in relations, each with several* attributes, *or*
2. a text *data model, where documents are represented as contiguous characters, typically interspersed with tags to capture their various logical, semantic, and presentational features and relationships*

Each approach has its own strengths and limitations. In our work, we study how a hybrid system based on a combined text/relational model can support document management. We describe database design trade-offs involving the appropriate placement of information in the text and relational database components. With an appropriate design, the advantages of both models can be exploited, while the shortcomings of using them individually are diminished.

We propose a set of primitive operations and a methodology for using them to evaluate the various alternatives for data placement. The methodology consists of simulating pre-defined, representative, document management tasks using the primitive operations and studying the numbers, types, and the time performance of the operations involved. Using some representative document management tasks as examples, we demonstrate the use of the methodology and the primitive operations to study and compare the processing of the tasks in the various data models mentioned above.

To the Glory of Our Lord, Jesus Christ

Without His providence and bountiful supply of wisdom and physical strength during the last five years, computing studies would have never been a possibility to me, let alone reaching the Master's level.

On a human level, my greatest thanks and heartfelt gratitude are due to my supervisor, Professor Frank Tompa, whose guiding hand has led me away from pitfalls and dead-ends innumerable. In the past twenty months, his sagacious and timely direction and comments, as well as his encouragement, patience and support have made this research work so much fun and fruitful.

In addition, I express my deep appreciation of the tremendous and commendable effort that my readers, Professors Donald Cowan and Mariano Consens have expended on reading the drafts of this thesis and their numerous valuable comments and suggestions for improvement.

I also owe my gratitude to Dr. Ian Davis and Mrs. G. Elizabeth Blake of the Centre for New Oxford English Dictionary and Text Research at the University of Waterloo for providing me their generous and enthusiastic assistance in using the software and hardware necessary for the completion of my research work.

Last but not the least, financial support from the University of Waterloo, the OpenText Corporation, and the Natural Sciences and Engineering Research Council of Canada (under grants CRD147259 and OGP0009292) are gratefully acknowledged.

Contents

List of Tables

List of Figures

1 Introduction

1.1 The Research Problem and its Context

The research reported in this thesis operates within the context of an ongoing project [BCD+95] at the Centre for the New Oxford English Dictionary and Text Research of the University of Waterloo (hereinafter referred to as the Centre) to study various issues related to the design and use of text databases. In that project, a prototype database management system has been implemented to support data manipulation and query processing in a combined text/relational data model. In that model, a document may be broken into different components and stored separately in some underlying text and/or relational databases. We have used that prototype for our preliminary research, to gain a better understanding of the functionality and operation of the combined text/relational data model.

In our research, we study how a document may be broken into components and how such components should be partitioned among the underlying text and/or relational engines.

We illustrate our research problem with a short example.

A Collection of Messages Consider, for example, the problem of representing and storing a collection of messages, such as the following :

> **To** : Prof. F. Tompa
>
> **From** : Kar Yan Ng
>
> **Date** : 10th September 1995
>
> *I enclose for your attention a draft of my thesis.*
>
> *The research reported in this thesis operates within the context of an ongoing project.*
>
> *I Look forward to receiving your comments on the draft at your earliest convenience.*
>
> *Regards,*

> **To** : Kar Yan Ng
>
> **From** : Prof. F. Tompa
>
> **Date** : 11th September 1995
>
> *Thanks for the draft. I'll give you my comments as soon as I've finished reading it.*

```
<collection>
<memo>
<receiver>Prof. F. Tompa</receiver>
<sender>Kar Yan Ng</sender>
<date>10th September 1995</date>
<body>
<paragraph>I enclose for your attention a draft of my thesis.</paragraph>
<paragraph>The research reported in this thesis operates within the context of an ongoing project.</paragraph>
<paragraph>I Look forward to receiving your comments on the draft at your earliest convenience.</paragraph>
<paragraph>Regards,</paragraph>
</body>
</memo>
<memo>
<receiver>Kar Yan Ng</receiver>
<sender>Prof. F. Tompa</sender>
<date>11th September 1995</date>
<body>
<paragraph>Thanks for the draft. I'll give you my comments as soon as I've finished reading it.</paragraph>
</body>
</memo>
</collection>
```

Figure 1: Storing Messages in a Text Model

Storing Messages in a Pure Text Model The above collection of messages can be viewed as a simple text in an ASCII form as shown above. In this form, it is difficult to provide efficient access without better management of its structure. Therefore, several researchers have proposed the use of SGML [Int86] to represent the logical structures contained in the texts. Structural tags are embedded within the text to delimit individual document components. For instance, the above messages could be represented using structural tags as shown in Figure 1.

It may be noted that the entire collection is enclosed within a pair of `<collection>` tags and may be stored within a single contiguous file. To facilitate access to the structured text, indexing may be performed in which the positions and extents of each document component, the words, and even the characters are recorded in some data structures ([ST93] and [CCB95]).

The Pure Relational Model Alternatively, the document skeleton could be stored in a traditional database, for example, one based on the relational model. In this case, one representation could be as

receiver	*sender*	*date*	*body*
Prof. F. Tompa	Kar Yan Ng	10th Sept 1995	`<paragraph>I enclose for your attention a draft of my thesis.</paragraph> <paragraph>The research reported in this thesis operates within the context of an ongoing project</paragraph> <paragraph>I Look forward to receiving your comments on the draft at your earliest convenience.</paragraph> <paragraph>Regards,</paragraph>`
Kar Yan Ng	Prof. F. Tompa	11th Sept 1995	`<paragraph>Thanks for the draft.  I'll give you my comments as soon as I've finished reading it.</paragraph>`

Table 1: Storing Messages in a Relational Table

receiver	*sender*	*date*	*memo_id*
Prof. F. Tompa	Kar Yan Ng	10th Sept 1995	1
Kar Yan Ng	Prof. F. Tompa	11th Sept 1995	2

Table 2: Message Headers

shown in Table 1. Notice that the `body` column is of type `string` or `varchar` which may be composed of multiple paragraphs, but such substructures are not represented in the database schema underlying Table 1.

To represent the paragraphs explicitly, it is better to encode the document structures and contents in two tables as shown in Tables 2 and 3. For each message we have factored out the document components which are common to the whole message and place them into a Message Header Table (Table 2) whereas the message bodies are put in a separate Message Bodies Table (Table 3). The two tables are connected by the identifiers under attribute `memo_id`.

To facilitate access to the tables, indices may be constructed. But unlike those indices used in the text models, typically indices in the relational models are built on entire column entries rather than being

memo_id	*para_#*	*paragraph*
1	1	`I enclose for your attention` `a draft of my thesis.`

1	2	`The research reported in this`
		`thesis operates within the context`
		`of an ongoing project`

1	3	`I Look forward to receiving your comments` `on the draft at your earliest convenience.`
1	4	`Regards.`

2	1	`Thanks for the draft.  I'll give`
		`you my comments as soon as I've`
		`finished reading it.`

Table 3: Message Bodies

character- or word- based.

The Combined Text/Relational Model In this project, we examine the use of a combined text/relational model to support document management.

As in the pure relational model described above, we may factor out all the document components containing no substructure into separate relational table(s). On the other hand, instead of putting the various other text components in some other relational tables, in the combined model we put these texts into a contiguous file as in the pure text model.

To continue with our message collection example, the header information may, as in the relational model, be placed in a Message Header Table (Table 2). However, the textual contents in the message bodies are kept in a single file as shown in Figure 2 :

Similar to the pure relational model, the relational header table and the components `paragraphs` contained in the text file are connected by identifiers. In the combined model however, the paragraph ordering is implicit in the text representation and therefore it need not be explicitly recorded. It may be noted that the `memo_ids` have been stored with the `paragraph` components so that the text management

```
<messagebody>
<memo><memo_id>1</memo_id>
<paragraph>I enclose for your attention a draft of my thesis.</paragraph>
<paragraph>The research reported in this thesis operates within the context of an ongoing project.</paragraph>
<paragraph>I Look forward to receiving your comments on the draft at your earliest convenience.</paragraph>
<paragraph>Regards,</paragraph>
</memo>
<memo><memo_id>2</memo_id>
<paragraph>Thanks for the draft. I'll give you my comments as soon as I've finished reading it.</paragraph>
</memo>
</messagebody>
```

Figure 2: Storing Message Body in a Combined Text/Relational Model

system may use them to support linking to the rest of the data.

As in the pure text model, we may construct indices on the file containing the message bodies to facilitate access to its contents and document components. Indices may also be built for the relational columns as well, as in the pure relational models described above.

In this thesis work, we identify some important criteria by which we may decide on the appropriate partitioning of data between the text and relational components of such a combined data model with the goal of achieving better efficiency in data manipulation.

1.2 Data Placement in Federated Databases

A *federated* database management system is a type of distributed database management system in which each constituent database management system is an independent and autonomous centralized system that has its own local users, transactions and administration [EN89]. Specifically, in this section we examine some major issues in determining the appropriate data placement in a federated database environment that contains both text and relational sub-systems. Following the terminology used in [BCD+95], in this thesis we call this particular type of federated database managements the *combined* text/relational database management systems.

Data Granularization Many document *entities* are related hierarchically within some *logical* structures. For these hierarchically-related entities, we need to decide on the smallest entities that could independently constitute a *granule* of information. Information units smaller than the granules selected may exist, but this would not be represented in *schema* or *grammar* which describes the structure of the *relational* or the *text* databases. Such small information units might only possibly be understood and manipulable by either the document management systems, the application programmes, or the users themselves (see also data *partitioning* below).

We have to decide on an appropriate level of granularity for both the *text* and *relational* databases.

The level of granularity is subject to some lower bounds as determined by the minimum logical and semantic components that are meaningful. Such lower bounds are different for different document types and the ultimate decisions are the database designers'. One possible lower bound is the individual fields (for tabular documents) or the tagged *regions* (for structured texts). For European languages at least, the absolute lower bound is probably a character.

Data Partitioning In addition, in our use of the combined *text/relational* data model to support a document management system, we need to decide on the following aspects :

1. which *granule(s)* of information should be stored in the *relational* database
2. which *granule(s)* of information should be in the *text* database.
3. whether and what *type* of information (e.g. the relational schema, document grammar) should be stored in the *database management system* managing the combined *text/relational* database.
4. what information should be encoded in the *document management system* sitting on top of the combined *text/relational* database
5. what information should be left to be encoded implicitly in the *application programmes* using the document management system,
6. finally, what information should be left to be stored in the heads of the human users. To wit, how much knowledge must the human users to learn in order to manipulate the document management system properly and use it to its full capability.

For the purpose of this project, we are primarily interested in the first two categories, even though we may touch on the others in passing.

Our literature survey ([CDY95], [CP84], [Chu92], [CMVN92], [ÖV91], [YMW$^+$85], [MIMH85], [CP87], [NCWD84], [BG92], and [NM89]) reveals the following main considerations for deciding on the appropriate data partitioning for a given federated database. The overriding objective is to minimize the inter-site communication time and maximize the proportion of local data accesses.

1. *Maximizing Local Processing*

   To minimize the need to perform *join*s over data stored in different sites, data likely to be accessed together should be placed closely together in the same relation in the same site as far as possible. Empirical findings of query types and frequency, plus the semantic relationships among data entities are two complementary bases on which to assess the likelihood of the information of different entities being accessed together.

2. *Maximizing Parallelism of Query Processing*

   A higher degree of self-sufficiency of each constituent database allows query processing to be performed locally as much as possible and the databases can work with higher parallelism with one another.

3. *Minimizing Data Redundancy and Update Consistency*

   A certain amount of data redundancy due to replication may be unavoidable in order to increase the amount of local query processing and to maximize concurrency of query processing in different sites. However, high data redundancy leads to higher total cost in storage space as well as a large amount of updating work in order to maintain consistency of all copies of the same data in the system. In this regard, it is desirable to make sure that overlapping attributes are those that are seldom modified.

However, while these issues are of significant relevance to the *data partitioning* issue in a distributed database environment in general, they do not explicitly take care of the fact that query processing speeds may be significantly different in the different constituent databases. In our particular context, owing to the significantly different schemata and component matching capability in a relational database management system and its text counterparts, there may be good reasons in many situations to forgo maximizing local query processing and parallelism in order to attain a better overall query processing performance.

Data Organization Last but not least, after we know how to *granularize* and *partition* the document information, we still have to determine how to organize the information in each of the respective sub-systems in which they reside.

1.3 Contents Overview

In Chapter 2, we examine some basic features of a document and its structures. We examine the various major document management functions and their functional requirements, with particular reference to the retrieval and storage aspects.

In Chapter 3, we define a set of primitive document manipulation operators. We intend that this set of operators form the basis of our discussion in the subsequent chapters. We demonstrate the usefulness of the operation set along the dimensions of *orthogonality*, *completeness* and *measurability*.

In Chapter 4, we review several major data models commonly used for document databases. We show how to convert document representations from one model to another using our set of primitive document operators defined in Chapter 3. We also examine some approaches by which the inter-relationships among entities residing in databases with different data models may be represented.

In Chapter 5, we examine the various document management operations that are performed in text and relational models. Insofar as they are related to the retrieval aspects, we attempt to express them in terms of the primitive operations we have defined in Chapter 3. We discuss how the types, number, and time performance of those primitive operations could be determined by the data models used by the underlying databases and the way data is represented and partitioned amongst them.

In Chapter 6, we conclude by giving a methodology for using the primitive operations to evaluate data placement alternatives in a federated database environment supported by a combined text/relational model. In addition, we propose some possible directions for future research.

2 Document Management : Tasks and Requirements

2.1 What are *Documents* ?

The *Oxford English Dictionary* [SW89] defines a *document* as follows :

> *Something written, inscribed, etc. which furnishes evidence or information upon any subject, as a manuscript, title-deed, tombstone, coin, picture, etc.*

In the field of computing and data processing, we may take the term *document* to mean a collection of recorded information. Such a collection may exist in some permanent forms, e.g. being stored as a file or files on a disk, or be generated at run-time from a collection of components that exist in some permanent forms.

With the advent of hypertext [Nel87] and World Wide Web [Ber93] technologies, the boundary of what comprises a *document* becomes open. In a sense, it can be said that all the information currently represented in electronic format is contained in one '*global document*'. A *document* may no longer be a *static* and *bounded* entity. This underscores the need for document representations that can support retrieval, sharing, and update efficiently.

2.2 Document Structures

In this section, we overview the three main types of document structures commonly used to model documents and how they interact. Of these, the *logical* structure is the main focus of our thesis. Using a technical report document type as an example, in Figure 3 we give an illustration of the three document structures to be discussed in the following subsections and their inter-relationships.

2.2.1 Logical Structures

These capture the *syntactical* organization of a document. As such these are tools for representing human knowledge in textual forms. The logical structure is the device by which an author organizes thoughts and presents them to readers in a logical manner.

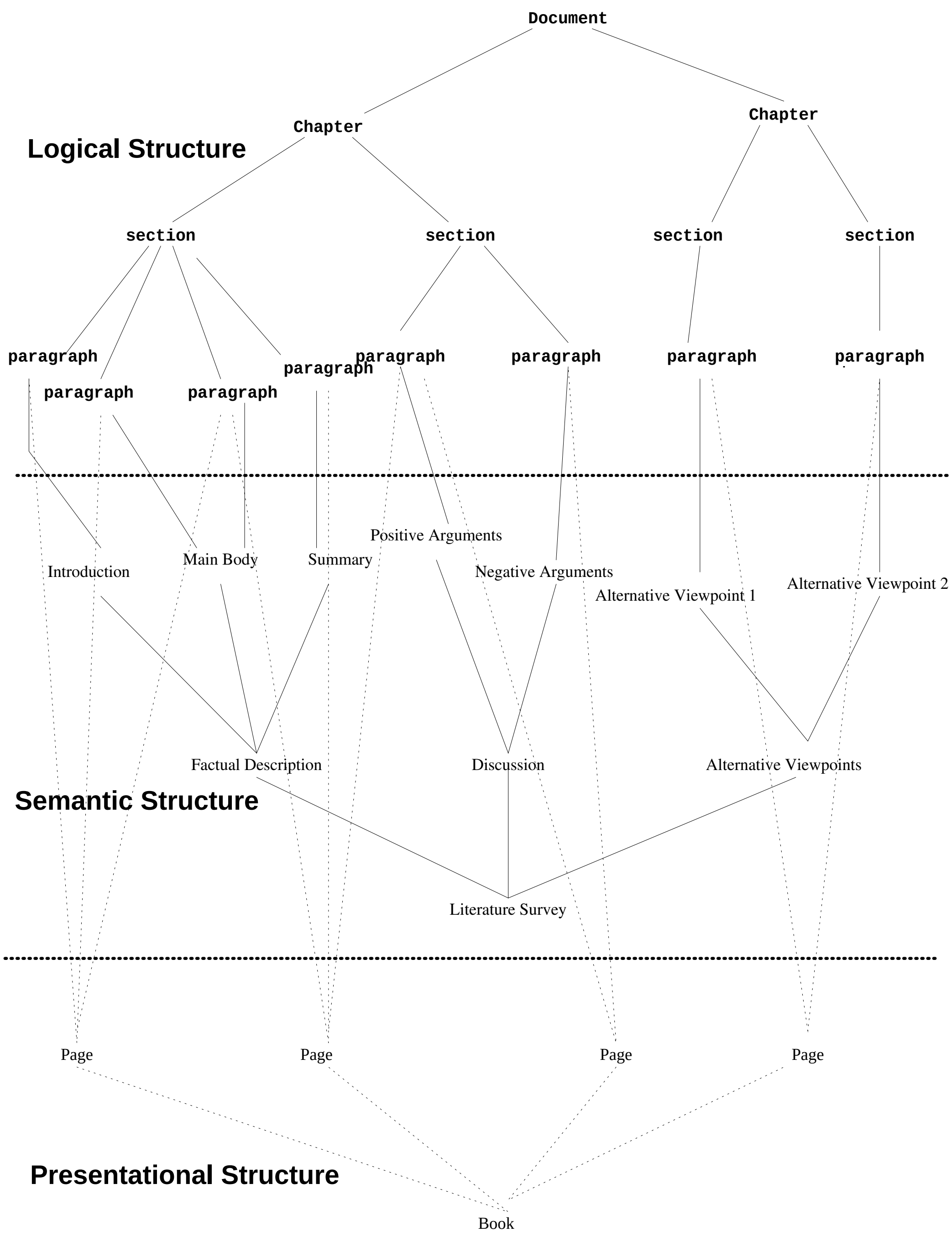

Figure 3: An Example of The Three Document Structures of a Document and their Interrelationships

2.2.2 Presentational Structures

These describe how a document is to be formatted and presented. Typical *presentational components* in a technical report document type include :

1. *page*
2. *line*
3. *column*

2.2.3 Semantic Structures

The *semantic* structures convey the organization of the ideas presented in a document. In principle, the *semantic* structure is supposed to model the document at a higher and more abstract level than that of the *logical* structure which tends to focus chiefly on the 'hardware' of a document like `paragraphs` and `sections`. See Figure 3.

In practice, we find it hard to clearly distinguish the *semantic* structure of a document from its *logical* counterpart. For example, in a well-written article, each `paragraph` contains one point of argument, and each `section` is typically used to convey ideas about a certain main theme of the matters under discussion. In fact, by putting things within a certain *logical* component, an author has already indicated that those things are closely related to one another in some aspects semantically.

Moreover, since they typically operate on a more abstract level, the semantic structures and their components are much harder to be clearly demarcated compared with the logical ones. A `paragraph` is clearly marked by white space, but where a particular argument or viewpoint starts or begins is much harder to ascertain.

In fact, we severely doubt whether a hierachical model is the right tool to capture the *semantic* structures of a document [RTW93]. Probably we need modelling apparatus which has been specially developed for modelling at such a level of abstraction. One possibility could be *first* and *second order* logics used in *deductive databases* (e.g. see [RSB$^+$87]).

Therefore, for the purpose of our thesis, we concentrate on the *logical* structure. In fact, in many places we implicitly *assume* that it coincides with the *semantic* structures of the document under investigation.

2.3 What is Document Management ?

[WLL85] defines *document management* to be :

> *the document preparation, communication, and management aspects of office systems.*

We have found no formal and exact delineation of such aspects. Based on common usage and our literature review however, we consider that these include the following :

1. document representation
2. document manipulation
   (a) document retrieval
   (b) document display
   (c) document creation and editing
3. document control
   (a) document sharing
   (b) document security

In this thesis, we focus on the document representation and manipulation aspects of document management. In addition, we extend the scope of document management to cover those document-related activities outside an office environment as well.

In the following sections, we briefly outline and discuss the main functional requirements pertaining to each of the above-mentioned aspects of document management, with particular attention to their impact on the ways in which documents are to be represented at the logical level. This forms part of the groundwork on which to build our subsequent discussion on using data models for document databases.

2.4 Document Representation

Documents need to be stored in media that are reliable and durable. In addition, to economize on the cost of storage, we seek to minimize data redundancy in storage. However, this needs to be balanced against the need for efficient retrieval. In particular, in a distributed database environment, we need to strike a compromise between the amount of communication time among the databases on the one hand, and the availability of storage space, data redundancy for each local database, and local query processing time on the other. Such a compromise substantially determines how data placement and partitioning should be performed in a distributed database system.

Furthermore, we need to choose a data model having regard to the document types and the anticipated query types so that information loss is minimized. Almost invariably each document type has its own natural way of representation. For instance, tabular documents might lend themselves to the relational table representation more naturally than a document containing only plain text.

2.5 Document Manipulation

2.5.1 Document Retrieval

We want information contained in documents to be retrievable with a reasonable response time.

In the following paragraphs, we discuss some common approaches to document retrieval and their main functional requirements as far as data representation and modelling are concerned. We discuss *link resolution*, a specialized form of document retrieval. The efficiency of handling links could be significantly influenced by the data models chosen for supporting them.

Search for Document In document searches, we look for target documents that match our requirements as expressed in some querying languages.

Some popular examples of retrieval systems that support document searches include Wais [Lin92], Open Text [Ope95], Lycos [ML94], and Harvest [BDM$^+$94]. Users submit search queries to a document database front end, typically in the form of :

1. *keywords*,
2. *contexts* within which matches should occur
3. other document *attributes* like the names of authors, dates of writing, translation, and publication, publishers, etc.
4. some optional boolean operators to specify the desired relationships among the above items

The matched documents that satisfy the set of user-specified requirements (or some parts thereof) are returned, either

1. in their original forms, or
2. as represented by the addresses of the places where the target documents are stored. To promote *portability* of documents across document management systems, such addresses should follow some uniform or commonly used syntax. As a common example, the addresses may be in the form of *Uniform Resource Locators* (the URLs[1]) in an HTML document.

   Alongside the document addresses, the document retrieval system may display some short descriptions summarizing the contents of the documents [BDM+94]. Based on such information, the users can traverse the links and arrive at the source documents which may contain links for further traversals.

To facilitate this mode of retrieval, we need data models that allows the various document description items that are likely to be specified as search conditions to be searched efficiently having regard to their syntactic and semantic characteristics.

Link Resolution as a means to support Document Retrieval With the availability of fast tele-communication links and high-resolution monitors, the use of link resolution to support multimedia document searches has quickly risen in popularity in recent years. Typically users are presented with a start-up page[2] containing a number of links. Such links may be accompanied by strings or images/icons that convey a brief outline of the contents of their targets.

Links carry information on how to locate the target documents. During link resolution, such information is interpreted by the document management system to retrieve the document.

Various mechanisms of link resolution have been proposed, typically falling into two main categories : *static* and *dynamic* resolution.

In **static resolution**, the addresses of the link *targets* are hardcoded in the page (typically in HTML [Ber93] format) facing the users. Links are followed by the document management system executing the appropriate document transfer protocols (e.g. `http`, `gopher`, `ftp` for documents located at other machines) and database retrieval mechanisms (at local machines) to obtain the required documents.

[1]see `http://www.w3.org/hypertext/WWW/Addressing/URL/URI_Overview.html`
[2]the start-up page could be either pre-set or generated by an initial query

In **dynamic resolution**, we add one additional level of indirection to link addressing. Instead of hardcoding the actual target address in a document page, we encode the mechanism to locate that address. Such mechanisms could be some keywords or a predefined query which the document management system could interpret and invoke the appropriate document retrieval operations accordingly. Some examples of such dynamic link resolution are found in the use of CGI (the *Common Gateway Interface*) scripts, [Hea92], [Car94], and [BTR93].

In the case of CGI scripts used in HTML documents, the clicking of a link triggers the execution of some pre-written scripts which, based on some predefined *keywords* associated with the link *anchors*, then execute the necessary information retrieval operations either locally or in some remote sites.

In the project Microcosm (see [Hea92] and [Car94]) at the University of Southampton, researchers place all link information (such as the *target* addresses and the identifier of the links) in a *link* base separated from the document database. Only the link identifiers are kept in the documents. Modification of the *target* addresses necessitates only modifying the *link* bases. Link traversal involves looking up the *link* information in the *link base* based on the *link identifiers.* Based on the *target* addresses returned from the *link base*, the target documents may then be fetched.

On the other hand, a *dynamic resolution* prototype has been built to operate on the OED (short for the *Oxford English Dictionary* [SW89]) database at the University of Waterloo [BTR93]. Link traversal involves searching for some keywords in the OED database and returning the appropriate targets as results to be displayed to the users. In addition, [BTR93] demonstrates another advantage of *dynamic resolution.* It permits different link resolution mechanisms to be invoked based on

1. the context in which the link traversal is triggered and
2. the applications handling the link resolution.

2.5.2 Document Display

For the information contained in a document to be usable by its human users, its contents must be rendered in a form in a medium perceptible by one or more of the human senses. We focus on the visual sense for the discussion in this thesis.

From a database designer's perspective, we are primarily interested in ensuring that the information retrieved from the database may be efficiently organized into a displayable format. More specifically, we need to form a string out of the retrieved contents which is then written to a specified display device. Users may be allowed to control the formation process of the display string by specifying their display requirements using some language devices. The document management system then sees to it that such display specifications are duly reflected and followed in forming the display string.

2.5.3 Document Creation and Modification

One main purpose of document creation and its subsequent modification is storing and communicating knowledge. In creating a document, an author endeavours to encode his/her ideas in the structures and contents of the document, in the hope that ideas can exist in a more durable form and that the readers at the other end of the communication channel can get hold of the author's ideas by following a reverse process to decode the structures and contents of the document.

The task of the document management system is to package a piece of text into a set of tuples or text fragments according to a specified schema to be inserted into the appropriate database with or without replacing or deleting the existing document components there. In order to do this, the system must first parse a piece of text input or edited by users to identify and possibly isolate the structure in the text.

In addition, at the underlying databases, document creation or modification action would cause the necessary modification to the data and indicies.

2.6 Document Control

2.6.1 Document Sharing and Reuse

Traditionally (and even with the prevalence of WWW and hypertext), most documents are stored at their local sites as single files in many major document processing formats such as WordPerfect [Wor90] and Microsoft Word [Mic94a]. It has been the users' responsibility to break their documents into components if they so wish. Some document processing systems have facilities to allow their users to specify how a document is to be composed at run-time. One example would be the `include`[3] facility of LaTeX[Lam85].

One major shortcoming of this approach is that it severely restricts the *reusability* and *sharability* of a document. It makes configuration management of the document very inefficient. Versioning can only take place at the level of entire documents. To permit some parts of a document to be revised, reused or shared among a team of people, the document owners either have to allow others access to the whole documents (in which case simultaneous modification of different parts of the same document by different users could not be allowed) or do substantial editing (e.g. cutting and pasting) to extract the necessary components and put them into separate files.

As far as the document management system is concerned, one major requirement for document sharing and reuse is that the system has to be able to handle sub-units of documents efficiently. In addition, it must allow the location of such sub-units in a document to be specifiable so that a complete document can be reassembled on demand. With such functionality, document sharing and reuse among multiple users may be well-supported.

2.6.2 Document Security

The objective here is to make sure that a piece of document is *available* in an *uncorrupted* form, only to the *users* for whom the document is intended *but* not to others. According to [IES94],

> *Security in relation to electronic documents means maintaining their availability, integrity and confidentiality by minimizing the risk of loss, corruption, and unauthorized access.*

As far as database design is concerned, we want to make sure that each document component is associated with the right security specification for its intended users.

[3] even though only 1 level of nesting is allowed.

3 Basic Document Management Operations

3.1 Objective and Scope

In this chapter, we define a number of basic document management operations to help characterize the operations used to support the retrieval and storage functionalities of a document management system. Our objective of designing such an operation set is to use the occurrence and timings of their member operations to guide data placement decisions in a federated database system involving both text and relational data models, to achieve better time performance in retrieval and storage activities.

In defining this list of basic operations, we intend them to be high-level abstractions of the query processing activities going on in a document database to retrieve and store documents. We have endeavoured to make the operations as independent of implementation and data model as possible.

In the following sections in this chapter, we start by outlining what query processing activities each of these operations represents. For each operation, we state some factors which would significantly affect the time performance of the respective operations. We will discuss the relevance and significance of such performance factors in Chapter 5 where we discuss and illustrate :

1. how the main retrieval and storage functionality used in a document management system in its various document management tasks, as identified in chapter 2, may be implemented using these basic operations,
2. how the time efficiency with which these basic operations are performed may be influenced by the data models chosen to represent the documents and by the ways data are allocated among the constituent components of the data model chosen, and
3. how such basic operations may be used to guide the data placement and query optimization decision-making of a combined text-relational database designer.

Following the definition of the operation set, we informally evaluate it in the dimensions of *consistency*, *orthogonality*, *completeness*, and *measurability*. We do not attempt to formalize our definition of these operations and therefore we cannot give a formal proof that the operations satisfy each of these criteria.

In all the discussion in this chapter, we use the term *text component* to refer to a document component plus the structural tags in which it is enclosed. In addition, we use the term *collection unit* to mean a *base* relation in a relational model; analogously, in a text model, we take a *collection unit* to be a text object that is not contained within another text object. One example of this would be a `collection` in Figure 1.

3.2 Retrieving Sets of Document Components

3.2.1 Locating Components

locateComp (set of component contents or locations,
    grammar or schema of the set,
    names of `components` to be located in the set,
    `select` condition)
    return set of component locations

The set of component contents may be either a set of tuples or a set of text fragments. Alternatively, the first argument could be a set of component locations which could be the locations of either texts or tuples.

The location set, both in input and output, consists of pairs in the form of : (`<component name>`, `<location>`), where the locations could be disk locations or other equivalent, indirect, forms of addressing.

In this operation, we find all the locations of the places in the given set of input tuples or texts as constrained by any given schema or grammar, where the specified selection condition is satisfied.

The tuple and text inputs to the operation may come from either base or intermediate relations or texts.

To meet the *relational completeness* requirement for *selection* (see subsection 3.7.3 below) for the basic operation set, we require that the *select* condition may contain a formula which supports the use of :

1. constant expression operands
2. component identifier operands

3. regular expression specifications of string operands

   The component identifiers may be the field names (or generic identifiers) in an SGML document or attribute names in a relation.

4. arithmetic comparison operators $<$, $=$, $>$, $\leq$, $\neq$, and $\geq$

In addition, to give the locateComp operation some text manipulation functionality, we require that the *select* condition supports the text position comparison operators *proximity* and *containment*. For example, PAT's operators `fby`, `near`, `including`, and `within` [ST93] could be suitable operators for this purpose.

The execution time attributed to this operation includes time spent in performing any index lookup and text searching, pattern matching, and returning the locations of tuples, text components, or parts thereof, for which the conditions of selection are satisfied. In addition, it also includes the time spent on format conversion, e.g. between a text and string, and between a string and an integer.

An example of using the locateComp operation on Table 1 would be :

> locateComp (Table 1, schema of Table 1, { `receiver`, `sender` }, { `body` including "finished" })

for which the output could be { ((Table 1).`receiver`, pointer to the receiver field in tuple 2), ((Table 1).`sender`, pointer to the sender field in tuple 2) }

Based on the operation inputs, we anticipate the following to be significant performance factors :

1. total amount of data against which actual matches need to be performed
2. cost of performing a single (complete) match
3. selectivity of the condition (if any)

$$= \frac{number\ of\ components\ satisfying\ the\ condition}{size\ of\ input\ set}$$

3.2.2 Extracting Components

> extractComp (set of component locations)

return set of extracted components

This operation takes as input a location set such as that returned from the locateComp operation. Each component extracts the relational or text components belonging to the same *collection units* whose names and addresses of matches are contained in the input location set. The output is a set of pairs in the form of (`<component location>`, `<content>`). An example of performing extractComp on Table 1 using the output of the result of the previous locateComp operation would be

> extractComp ({ ((Table 1).`receiver`, pointer to the receiver field in tuple 2), ((Table 1).`sender`, pointer to the sender field in tuple 2) })

for which the output would be { (pointer to the receiver field in tuple 2, "Prof. F. Tompa"), (pointer to the sender field in tuple 2, "Kar Yan Ng") }

Based on the inputs, we anticipate the following to be significant performance factors :

1. size of the location set
2. cost of extracting one component

3.3 Component Transformation

Parsing

> parse (string,
>
> grammar of the text contained in the string,
>
> schema of the tuples or a set of text fragments to be formed)
>
> *return* sets of tuples or text fragments

In this operation, a string is parsed into its various components according to a grammar and the components are placed into some sets of tuples as specified by a schema. Alternatively, the tuples may be packaged into database insertion commands which may then be executed via a subsequent insert operation.

Assembly

> assemble (sets of tuples,
>> schema of tuples,
>> grammar of the text to be formed)
>> *return* a string

This is the inverse of the parse operation. It takes some sets of tuples and their schemata to produce a text according to a specified grammar by taking the components from each set of tuples in order and matching the component identifiers as appropriate.

One example of using the assemble operation would be :

> assemble ({Table 2, Table 3 }, schemata of Tables 2 and 3, grammar of the collection of message in Figure 1)

for which the output would be a string such as shown in Figure 1.

Based on the inputs, we anticipate the following to be significant performance factors for both parsing and assembly :

1. number of the tuple sets involved
2. total cardinality of tuple sets
3. cost of parsing or creating one assembled component

3.4 Component Modification

Insertion

> insert (contents,
>> collection unit)
>> *return nil* % side effect of writing `contents` to the *collection unit*

This operation inserts the input contents to the specified *collection units*. We assume that insertion is performed either locally or close to the machine running the database management system. (For remote storage, any time spent in the communication links will be accommodated by an additional operation comm to be discussed in the next subsection). Each component written is assumed to match a base relation tuple or a text object, and thus no byte offset needs be specified. In addition, this includes any necessary index creation and/or modification work. It also includes the time for any necessary rewriting of parts or whole of the databases. The timing would also include the time spent in waiting for locks, disk access, and any queueing in the underlying operating systems.

To allow for the physical relocations of document components, we do not propose to use physical disk locations as parameters. Instead, we simply specify the logical location of the destination storage component and let the underlying database management system decide where to place it.

Moreover, specifying the destination storage component is not necessary if that information has been embedded in the content to be written. For instance, the contents to be written may be the insertion commands to be executed at the underlying databases.

One example of using this operation would be :

insert (contents of tuple 1, Table 1)

Based on the inputs, the significant performance factors are the size of contents to be written, the size of the existing databases, and any related access structures, and the number and type of indices to be updated.

Deletion

delete (locations of components)

return nil% the side effect of deleting the components as identified by the input locations

This takes in a set of component locations returned from the locateComp operation and delete the existing contents there in the database. All of the locations must be in the same *collection unit.*

One example of using this operation would be :

delete (pointer to the sender field in tuple 1 in Table 1)

Based on the inputs, the significant performance factors are the size of component to be deleted, the size of the existing databases, and any related access structures, and the number and type of indices to be modified.

3.5 Communication

comm (site 1,

site 2,

message contents)

return nil % side effect of transmitting `message` from `site 1` to `site 2`

This operation moves a piece of data from one site to another. We take this to include the time spent in the communication link, including any waiting and re-transmission times.

Based on the inputs, two significant performance factors are :

1. size of the message
2. speed and length of the communication links between sites 1 and 2

3.6 Operations on Tuple Sets

3.6.1 Joining

join (location set 1,

location set 2,

`join` conditions,

schemata for the input and output sets)

return a set of component locations for which the contents they point to satisfy the `join` condition

The *join* condition follows the same definition as the *select* conditions for the locateComp operation as described in subsection 3.2.1 above, except that the join condition does not allow the use of text position comparison operators. The input sets contain the locations of the tuples or texts to be operated upon by the join operation.

In either case, the input set may correspond to a collection unit or it may be generated from operations that return set(s) of component locations e.g. locateComp.

Like locateComp, the time for join includes the time spent on such format conversion work as those between text and string, and between string and integer.

Based on the inputs, three significant performance factors are :

1. sizes of the tuple sets
2. selectivity of the join conditions
3. cost of producing one joined tuple

3.6.2 Sorting and Set operations

sort (a set of text or tuple locations,
  sorting key(s),
  order,
  schemata for the members of the input set)
  return a set whose members have been sorted according to the specified sorting key(s) in the specified order

union (location set 1,
  location set 2)
  return the union of the two input sets

intersect (location set 1,
  location set 2)
  return the intersection of the two input sets

difference (location set 1,
  location set 2)
  return the difference of the 2 input sets

In all cases, the input set has to correspond to a collection unit or it may be generated from operations that return set(s) of component locations e.g. locateComp.

Based on the inputs, significant performance factors for these operations are :

1. the size of input set,
2. the selectivity of the operation (equal to 1 for sort)

$$= \frac{number\ of\ input\ components}{number\ of\ output\ components}$$

3. cost of producing one output component

In addition, for the sort operation we need to consider the timing for key comparison as well.

3.7 Usefulness of the Operation Set

3.7.1 Consistency

We examine the consistency of our operation set by studying how the input of each operation may be produced.

1. locateComp

   The sets of tuples or texts could be base relations or texts stored in the databases being operated upon. Alternatively, they may be produced by other operations : extractComp, parse, join, sort, and the various set operations.

   The name of components and the select conditions need to be input by the users or the application programming calling the locateComp operations based on user inputs.

2. extractComp

   The set of component locations is returned from the locateComp operations.

3. parse

   The string to be parsed may either be created and input by the users through editing or be retrieved from a database through the extractComp operation.

4. assemble

   The schemata and grammar are specified by the users or the application programmes. The set of tuples to be assembled is either directly input by the users or produced from the extractComp operation.

5. insert and comm

   The destination components for insert, and the communicating sites for comm are specified by the user or application programmes. The contents can be anything and as such outputs from any one of the operators.

6. delete

   The location sets could be generated by either the locateComp, join, sort, or the *set* operations.

7. join, sort and *set* operations

The list of sets could be formed from sets returned from any of the operators that produce location sets as their outputs. These include all the *set*, sort, locateComp, and join operations.

3.7.2 Orthogonality

By the construction of the operation set, we have made sure that one operation cannot be simulated by another and their functionalities do not overlap, with the exception of the join operation with the locateComp and extractComp operations.

Violation of *orthogonality* occurs in the above-mentioned case to the extent that join may involve component locating and extracting. To avoid such violation, we would have had to break the join operation into smaller operations. But this would severely reduce the *measurability* of the resulting operations since for most database management systems, their execution plans or sets of query commands do not support such a fine granularity of operations. In other words, without access to the source codes, it may not be feasible to take time measurement for the operations.

Therefore, as a compromise between *orthogonality* and *measurability*, we choose to keep the join operations alongside the locateComp and extractComp operations. In studying the number of basic operations used to perform a document management task, we have to take care to avoid double-counting.

3.7.3 Completeness

Starting with a discussion of the meaning of *completeness* in general, in this subsection we demonstrate the extent of completeness of the basic operation set we have proposed above.

We typically prove the completeness of a language by showing that it has at least the same expressive power of another one. But we are unable to do this here, because there is as yet no widely accepted standard language in which all document manipulation functionalities are supported.

Instead, we informally demonstrate the extent of completeness of our operation set by demonstrating that it is *relationally complete* and, by way of example tasks in Chapter 5, that the typical document management tasks as described in Chapter 2 may be simulated by our basic operations in the various data models being studied.

Definition of Relational Completeness Ullman [Ull82] defines a language to be *relational completeness* if it :

> *can (at least) simulate tuple calculus, or equivalently, relational algebra, or domain calculus.*

In addition, we extend the measure of *completeness* for our set of basic operators to cover some of the text manipulation functionalites provided by PAT [ST93].

Demonstration of Relational Completeness Ullman [Ull82] lists five basic operations that define *relational algebra* and we can simulate each one as follows :

1. Set *Union* and *Difference*

   We may achieve these by using our set operations union and difference respectively.

2. *Cartesian Product*

   This may be done by the join operation with no condition specified.

3. *Projection*

   This may be done by performing the locateComp operation on a tuple set to identify components and then issue extractComp to retrieve the required attributes from the tuple set. The input location set would be formed from pairs containing the required attribute names and the identifiers of the tuples.

4. *Selection*

   The *selection* operation in *relational algebra* is defined as follows [Ull82] :

   > *Let F be a* formula *involving*
   >
   > *(a) operands that are constants or component numbers*
   >
   > *(b) arithmetic comparison operators $<$, $=$, $>$, $\leq$, $\neq$, and $\geq$*
   >
   > *(c) logical operators $\wedge$, $\vee$, and $\neg$*
   >
   > *Then $\sigma_F(R)$ is the set of tuples t in R such that when, for all i, we substitute the i^{th} component of t for any occurrences of the number i in* formula *F, the* formula

F becomes true. For example, $\sigma_{2>3}(R)$ denotes the set of tuples in R whose second component exceeds its third component.

Suppose f is a subformula in F, S is the set of source tuples or texts upon which F is to be evaluated. Subformulas containing components belonging to the same sets of tuples/texts may be evaluated by one locateComp operation. Such subformulas may contain any arithmetic and logcial operators.

Let α_1 and α_2 be sub-formulas containing components belonging to different sets of tuples/texts such that $f = \alpha_1$ op α_2 . We may evaluate t_1 and t_2 individually using the locateComp operation.

(a) For op being $\vee$ or $\wedge$, we may obtain the value for f by performing a union or intersect operations respectively on the intermediate result sets of α_1 and α_2.

(b) For op in $\{ <, =, >, \leq, \neq, \geq \}$, we may perform a join operation between the intermediate sets of tuples/sets of α_1 and α_2.

(c) For f being $\neg\, \alpha$, we evaluate α using the locateComp operation and obtain the result for f by applying the difference operation to S and the intermediate result set from α.

3.7.4 Measurability

We study the *measurability* of the basic operations by examining the precision to which the time spent by each of these basic document manipulation operations may be measured if we are given a black-box database management system for which we are not permitted to modify its source code explicitly in order to obtain the timing for each of the operations to be measured.

As a general remark however, the precision and the reliability of all time measurements are limited by the precision of the timing statistics available from the database management system being studied. The timing statistics provided by Oracle 7 [Ora92a] for example are only accurate up the nearest $\frac{1}{100}$ seconds. Moreover, process timing is also significantly influenced by the workloads of the operating systems, hardware, and communication links at the time of measurement.

Moreover, in situations where we attempt to measure the time spent by an operation by observing the differences between the processing times of queries with and without the operation, special attention needs to be paid to the execution plans of the test queries. We must make sure that the queries with and without the operation are indeed executed in such a manner that the subject operation makes the only difference.

The sort **and** *set* **Operations** Their timings may be measured by the difference in query execution times between queries with and without the respective operations.

The component *transformation* **Operations** This may be estimated by studing the cpu times of the parse and assemble operations which are typically performed outside database management systems.

The comm **Operations** Apart from the sizes of the messages or contents to be communicated, the time performance of these operations is substantially determined by the size of the operating systems the existing databases and the hardware on which it operates. As such, timing for such operations may be performed outside the database management systems.

Timing information for comm may be measured by making a small application programme to send a message of known length from one machine to another via sockets.

The insert **Operations** Apart from the sizes of the messages or contents to be written, the time performance of this operation are substantially determined by the operating systems, the sizes of the existing database and its indices, and the hardware on which it operates.

Time spent by the operation may be measured by the time taken to insert a text or a tuple of known size into the given database(s). However, this might not reflect the full time cost of a modification operation in a database management system where merging of indices or database data are delayed.

The delete **Operations** Similar to the insert operation, the time spent by the operation may be measured by the time taken to delete a component of known size from the database. However, like the insert operation, this might not reflect the full time cost of the operation since garbage collection and consolidation of indices or data may be delayed.

The locateComp, extractComp, **and** join **Operations** In principle, the timings for such operations may also be measured by observing the differences in timing of queries with and without the operations. However, in practice, the applicability of this approach is constrained by the following system-dependent limitations :

1. For the first two operations, even though most database management systems may support their functionalities, the granularity of the corresponding query commands may not exactly correspond to our proposed operations. For instance, the locateComp and extractComp operations may be supported via a single `select` command and so it may not be practicable to measure their individual timings by submitting queries with and without the operations.
2. Queries with and without the operations may follow substantially different execution plans so that the time differences do not give a good or even meaningful indication of the timing for the operation being observed.

   For instance, for the join operation, queries with and without it may use different data structures and access paths.

In such situations, we propose the use of *regression analysis* techniques [Jai92] to identify the impact of individual operations. With this approach, we may use the total execution time for a query containing the operation as the dependent variable. The steps for executing the query, accessing the data, and storing the intermediate results are set out in an *execution plan* [EN89]. We use the execution plans supplied by the database management systems to identify the query processing operations that have been performed and the sizes of inputs to each of them. Suppose for simplicity we use a linear regression model. We may then use the sizes of inputs as independent variables and for each operation the regression coefficient would be its time cost per unit input.

To illustrate the approach, consider the following query submitted to Oracle 7.0[4] :

select * *from* `emp`, `dept` *where* `emp.dept.no` = `dept.deptno`
(‡)

The corresponding Oracle execution plan is shown in 4.

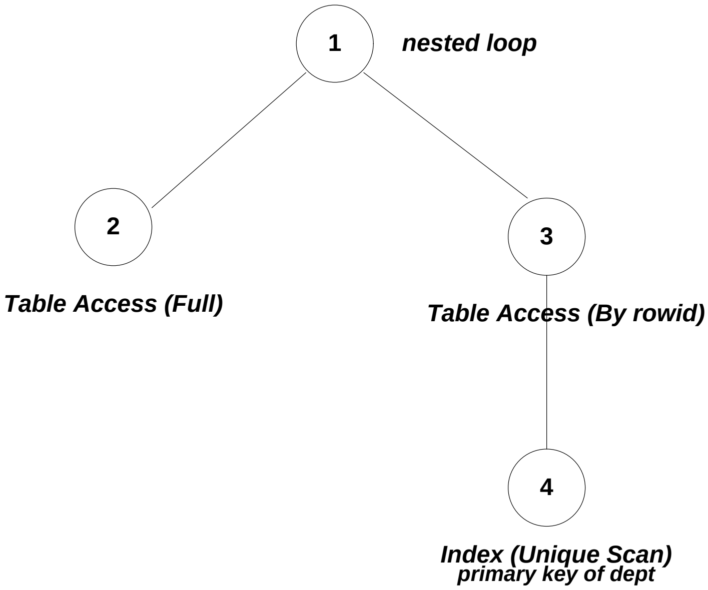

Figure 4: The Execution Plan of Query Example ‡

According to [Ora92b], the execution plan is interpreted as follows
:

1. `NESTED LOOPS` is an operation that accepts two sets of rows, an outer set and an inner set. Oracle compares each row of the outer set with each row of the inner set and returns those rows that satisfy a condition.

   Accordingly, this corresponds to the join operation in our basic operations set.

2. `TABLE ACCESS (BY ROWID)` is a retrieval of rows from a table based on its `ROWID`

[4]This example has been adapted from an example in page 13-56 of [Ora92c].

This corresponds to an extractComp operation.

3. `INDEX (UNIQUE SCAN)` is a retrieval of a single `ROWID`s from an index. This corresponds to an locateComp operation.

According to 13-57 of [Ora92c], Oracle performs the following steps to execute the example query :

1. step 2 accesses the outer table (`emp`) with a full table scan.
2. for each row returned by step 2, step 4 uses the `emp.deptno` value to perform a unique scan on the index of `dept.deptno`
3. step 3 uses the `rowid` from step 4 to locate the matching row in the inner table (`dept`)
4. each row returned by step 2 is combined with the matching row returned by step 4 and returns the result.

Expressed in terms of our primitive operations, this query would be processed as :

l_1 = locateComp (emp, schema of emp, *)

e_1 = extractComp (l_1)

l_2 = locateComp (dept, schema of dept, *)

e_2 = extractComp (l_2)

join (e_1, e_2, { (e_1.deptno = e_2.deptno) }, schemata of e_1 and e_2)

We are interested in knowing the times for the locateComp, extractComp, and join operations. Therefore, a linear regression model for the query execution time for such a query could be :

$$\begin{aligned} exec\_time &= join\_cost \times (size\ of\ e_1 + size\ of\ e_2) + \\ &\quad locateComp\_cost \times (size\ of\ \mathtt{emp} + size\ of\ \mathtt{dept}) + \\ &\quad extractComp\_cost \times (size\ of\ l_1 + size\ of\ l_2) \end{aligned}$$

where

1. $exec\_time$ is the query execution time as reported by Oracle
2. $join\_cost$ is the unit time cost for performing a the join operation
3. $locateComp\_cost$ is the unit time cost for performing an locateComp operation
4. $extractComp\_cost$ is the unit time cost for performing an extractComp operation

It is noted that using such a time measurement approach does require a good correspondence between our basic operations and the operations reported in the execution plans of the database management system being studied.

4 Data Models for Document Databases

4.1 Overview

According to [EN89], a *data model* is

> *a set of concepts that can be used to describe the structure of a database.*

Furthermore, [EN89] classifies *data models* as follows :

> **High-level** *or* **Conceptual** *data models provide concepts that are close to the way many users perceive data, whereas*
>
> **Low-level** *or* **Physical** *data models provide concepts that describe the details of how data is stored in the computer. Concepts provided by low-level data models are generally meant for computer specialists, not for typical end users.*
>
> *Between these two extremes is a class of* **Implementation** *data models, which provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer. Implementation data models hide some details of data storage but can be implemented on a computer system in a direct way.*

In the remainder of this chapter, we give an overview of four implementation data models for document databases. They have been selected for discussion primarily for their direct relevance to our thesis focus of investigating the data placement issue in the combined text/relational data model. At a generic level, we compare and contrast their functionality and performance. In the course of doing so, we would also pinpoint the main features, advantages and limitations of each data model.

There are relationships among documents and/or their components which are not explicitly represented in the structure of a data model as defined above. In a document, these relationships typically appear in the form of cross-references among document components. After the data model review, we discuss how such cross-referential relationships may be encoded within each of the implementation data models.

4.2 The Text Models

In the *text* model, a *document* is typically, though not necessarily, stored in its entirety as a single piece of text. If its structure is to be explicitly represented, we may do so by *marking up* the corresponding document components by some structural tags.

In these regards, the Standard Generalized Markup Language (SGML) [Int86] has been designed to provide a standard way of representing text documents marked up by structural tags. An SGML document typically contains a DTD (or the *document type definition*) which contains the grammar to which the text must conform and the text which is tagged according to the DTD. The tags are embedded within the text to define boundaries of logical blocks. Moreover, an SGML documents consists of elements whose structures and permissible contents are defined in the DTD. Each element is marked up by a pair of tags in the text. Each tag contains an identifier (the *generic id*) that identifies the element it represents and optionally some attributes with which that element is associated.

For instance, Figure 5 gives an example of how a report[5] containing a title and a chapter may be represented, where the newlines have been introduced solely for display purposes. The set of tags is predefined and their inter-relationship is represented in a grammar or a DTD as in Figure 7 . The textual contents lying between a pair of start and end tags (e.g. `<para>` and `</para>`) is called a *region*. Whereas we have defined *regions*, by explicitly placing tags around the text which they contain, in some text indexing implementations (e.g. [Ope93] and [CCB95]), we may instead define *regions* based on some user-specified *patterns* without explicit tag insertion.

A text database schema could then consist of :

1. the set of *tags*

   In the example in Figure 5, the set of *tags* is

   { `<report>, <title>, <chapter>, <intro>, <para>`
   }

2. optionally, the *grammar* (e.g. in the form of a DTD in SGML) describing the relationships among the tags.

Even without an explicit *grammar*, the document structure of a piece of text is implicit in the relative positions of the tags. As pointed

[5]extracted and adapted from the SGML92 report [AT92] and section 5.2 below

```
<report>
<title>Getting started with SGML</title>
<chapter shorttitle="challenge">
<title>The business challenge</title>
<intro>
<para>With the ever-changing and growing global market, companies and
large organizations are searching for ways to become more viable and
competitive.  Downsizing and other cost-cutting measures demand more
efficient use of corporate resources.  One very important resource is
an organization's information.</para>
<para>As part of the move toward integrated information management,
whole industries are developing and implementing standards for
exchanging technical information.  This report describes how one such
standard, the Standard Generalized Markup Language (SGML), works as
part of an overall information management strategy.</para>
</intro>
</chapter>
</report>
```

Figure 5: A Short Example Document Representation in the Text Model

```
<report>
<title>Getting started with SGML
<chapter shorttitle="challenge">
<title>The business challenge
<intro>
<para>With the ever-changing and growing global market, companies and
large organizations are searching for ways to become more viable and
competitive.  Downsizing and other cost-cutting measures demand more
efficient use of corporate resources.  One very important resource is
an organization's information.</para>
<para>As part of the move toward integrated information management,
whole industries are developing and implementing standards for
exchanging technical information.  This report describes how one such
standard, the Standard Generalized Markup Language (SGML), works as
part of an overall information management strategy.
```

Figure 6: A Short Example Document Representation in the Text Model without End Tags

out by [Tom89], in the absence of updates a grammar is superfluous under the assumption that start and end tags appears in the text as pairs conforming to a proper nesting as shown in the above example.

On the other hand, a grammar may have a *prescriptive* (as opposed to merely being *descriptive* of the existing structural state of a text) role as well [Tom89]. It tells its users what *should be* in addition to what *is* in a text. Without a grammar explicitly specified, no *data validation* can be performed when deletion, addition and other modifications actions are performed on a text.

Morevoer, the knowledge that the tags have to conform to a grammar permits many end tags to be dispensed with. As an illustration, consider the DTD in Figure 7, Figure 6 shows a modified version of the text shown in Figure 5 in which the structure can be fully determined.

Document Components may possess their own *attributes* as in the *relational* model. These may be represented either as *sub-components* inside the *components* they are describing (e.g. the `<chapter>` tags in Figure 6) or as *attributes* embedded with the corresponding structural tags (e.g. `shorttitle` in Figure 6).

```
<!ENTITY % text "(#PCDATA | emph)*">
<!ELEMENT report - o (title, chapter+)>
<!ELEMENT title  - o (%text;) >
<!ELEMENT chapter - o (title, intro?)>
<!ATTLIST   chapter
  shorttitle CDATA #IMPLIED>
<!ELEMENT intro - o (para | graphic)+>
<!ELEMENT  para  -  o (%text;)*>
```

Figure 7: The DTD of the Short Example Text with Optional End Tags

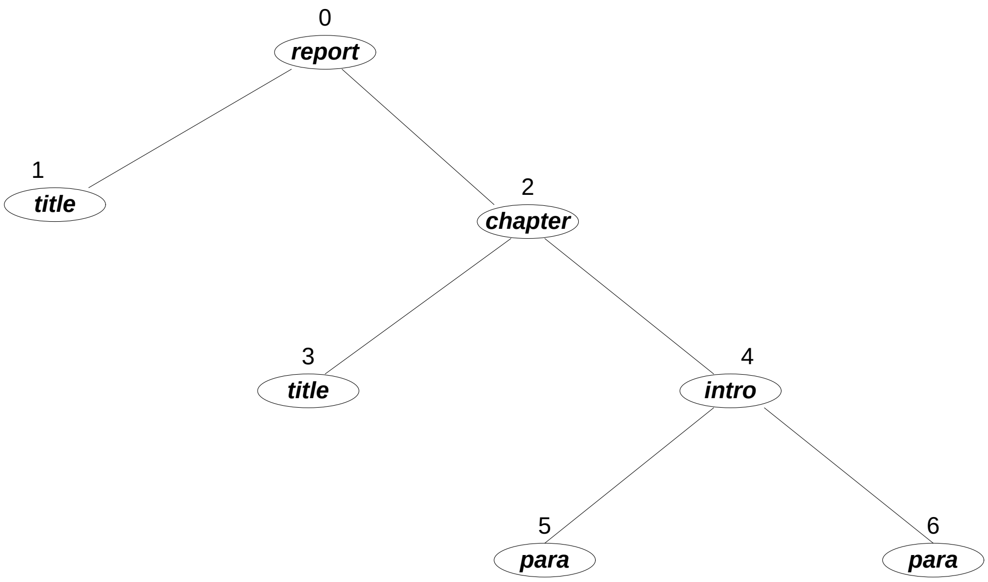

Figure 8: A Tree Representation of the DTD of the Short Example

4.3 The Grammar-specific Relational Models

In the *relational* model, a *document* is either :

1. stored in its entirety as an entry in a column of a relational table,
2. broken down into its various constituent components to be stored in *nested relations*, or
3. broken down into its various constituent components, each of which is stored as a column entry in a relational table

The relational data models to be discussed in this section share the common characteristic that the schemata they use are specific to the structure of the documents to be stored. For columns whose entries are represented by SGML or any other mark-up languages, we may store the corresponding DTDs or grammars in another relational table column. Other meta-data describing either the documents themselves or their components may also be represented as relational table entries.

In the following paragraphs, we discuss and contrast the above three approaches for using the grammar-specific relational data models to support documents.

4.4 Storing an Entire Document as a Relational Table Entry

This is the simplest approach. The entire document is treated as a long string of characters with any structure represented by embedded character string tags. To search for any patterns or document components in the text entry, string matching is the only alternative available, typically without the assistance of any indexing done on the texts in a preprocessing step.

A major and obvious shortcoming of this approach is that pattern matching and retrieval of document contents based on any document structures would be very inefficient. Essentially, to get the text that matches a pattern, would require matching and locating both the user-specified pattern(s) and the patterns that define the document components which are supposed to contain the input patterns.

The alternative approaches to be discussed below improve upon this rudimentary text storage schema by preprocessing the data so that the document structures are extracted once for all and recorded either inside or outside the database. This saves the time and effort of performing pattern matching work to locate the document components every time a query is posed, and it allows the update of subtexts in a straightforward manner. In fact, the same applies to any pattern matching over the text as well. There is a trade-off between the amount of pre-processing (typically indexing) work, and the amount of extra storage space (chiefly due to indexing, pointers, and other component marking devices) on the one hand, and the pattern matching and hence query response time on the other.

Another, more practical, limitation of this approach is that many database systems have placed restrictions on the maximum length of texts that can be held within one table entry. For ORACLE7 Server Release 7.0.15.4.0, there is a limit of 2 gigabytes [Ora88] for both binary and ASCII data.

In addition, as pointed out by [SAZ94],

> *One of the primary functions of a database schema is that of* validating *the data in the database.*

Without any knowledge of the structure of a document being stored, it is impossible for the database management system to handle this task.

On the other hand, an advantage of this approach is its storage efficiency. There is much less overhead for indexing, pointers, etc. Moreover, it is much simpler to assemble documents. The text contents in its original form, can be readily viewable and manipulatable using any viewing tools and editors. Manipulation can be performed by any application programme. As such, it is more portable as well.

4.5 Storing Document Components in Separate Tables

In this approach, document structures are recognized[6] and recorded implicitly in the relational schema.

To demonstrate the capability of this model to represent an SGML document, we show how the following DTD may be represented by the Entity-Relationships diagram in Figure 9 which may then be represented as Tables 4 and 5.

[6]either manually or through some automatic document recognition mechanisms as discussed in, for example, [FX93] and [Mar92].

sectid	shorttitle	title	intro

Table 4: section-title-intro

sect id	seq #	topic

Table 5: section-topic

```
<!ELEMENT section - o (title, intro?, topic*)>
<!ATTLIST   section
 shorttitle     CDATA           #IMPLIED
 sectid         ID              #IMPLIED>
```

Database managers have the discretion over how the document components should be distributed among the tables. At one extreme, we may have one table for each component. At the other extreme, all the components of a document may reside in one table. The decision is based on a compromise between the number of *joins* and the extra storage requirements due to the necessity of tuple *keys* in each table on the one hand and *data redundancy* and *data consistency* problem due to the presence of *data replication* on the other. The more tables, the more *joins* are required to link them back together, either at query time or during pre-processing in order to get at the required tuples. On the other hand, owing to the typically *nested* structures of many document types, the larger, outer, components[7] would typically need to be duplicated if they are to be stored in the same table as their deeply nested counterparts. Apart from their impact on storage costs, such data redundancy leads to larger tuple sizes and hence negatively affect the time performance of the locateComp and extractComp operations.

[7] with their *component ids* and possibly some textual contents (to the extent that such contents cannot be properly placed under some nested *components*) as well

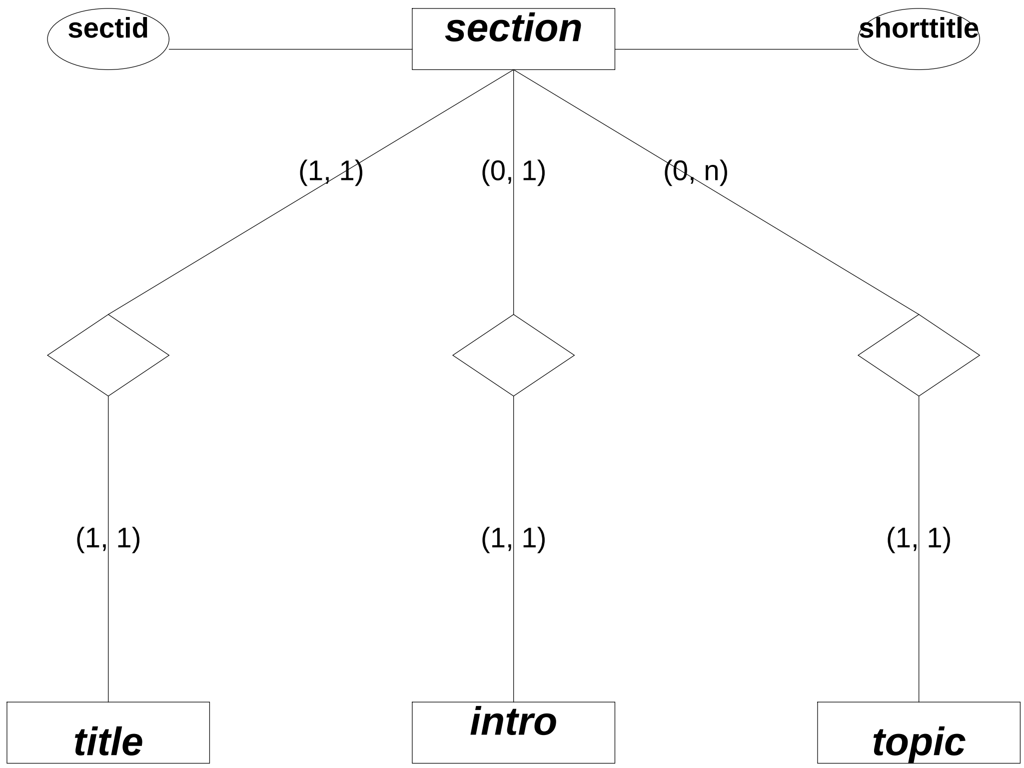

Figure 9: An Entity-Relationship Diagram for the `section` Element of a SGML92 Report

4.6 Storing Document Components as Nested Relations

To redress the above-mentioned shortcomings of storing documents in separate tables only to *join* them back when queries arise, researchers proposed a *nested* relational model [Tho93]. This model has the additional advantage of giving a more natural and intuitive representation of a document structure and as such make query formulation easier.

We illustrate below the nested relational schema using the example of our simplified bibliographic database :

```
report [
      report_id     INTEGER
      title         STRING,
      chapters      LIST of chapter,
] KEY = (report_id)

chapter [
      chapter_id    INTEGER
      title         STRING
      intros        intro
      is_chapter_in report
] KEY = (chapter_id)

intro  [
      intro_id      INTEGER
      paragraphs    LIST of paragraph
      is_intro_in   chapter
] KEY = (intro_id)

paragraph [
      para_id       INTEGER
      content       STRING
      is_paragraph_in intro
] KEY = (para_id)

report.chapter INVERSE OF chapter.is_chapter_in
chapter.intros INVERSE OF intro.is_intro_in
intro.paragraphs INVERSE OF paragraph.is_paragraph_in
```

In this example, many entities contain list of some other entities in the database. For instance, a `report` contains a *list* of `chapter` each of which may contain an `intro` which in turns contains a *list* of `paragraphs`. Morever, for each nested component there are inverse pointers pointing back to their nesting components so that for each entity we can find its immediate ancestors more efficiently.

In the words of [Tho93],

> *The idea of the nested relational model was developed around the need to extend the relational model to support* **complex objects** *as well as atomic attributes. This arose partly from semantic data modelling research which indicated that such structures are frequently required to model the real world, and partly from the need to efficiently implement relational databases at the physical level by supporting repeating groups.*

By nesting one inside another, we essentially have *joined* them together permanently, thus saving the time for tuple *joining* (for relations which have *nested* relationships) during query processing.

On the other hand, to properly handle nested[8] relations, we need additional operators in the *data manipulation language* (the DML) of the database management systems. We need to be able to *unnest* any *nested* relations that may be contained in tuples before other relational operators can work on them. Moreover, at the time of creating new table entries, we need to make sure that all their nested relations have been properly initialised and/or entered as well.

It may be noted that there is a strong resemblance between the nested relational models and the text models discussed above. The main reason is that a piece of structured text is analogous to a nested relation whose schema is defined by a grammar.

In this thesis however, we do not explore this model further, since this approach has not been significantly supported by commercial database management systems. Moreover, no standard has yet been accepted by either the industrial or academic community.

[8]or *non First Normal Form*, since *multivalued* and *composite* attributes are allowed

4.7 The Grammar-independent Relational Model

In a previous version of [BCD+95] published in the *Proceedings of the Advanced Database Conference 1994*, a type of relational model was described in which the schema is independent of the structure of the documents to be represented. To represent a general SGML document, the paper proposed three *virtual* tables as follows :

Text_nodes (`nodeid`, `genid`, `content`)

Text_attributes (`nodeid`, `attr`, `value`)

Text_structure (`a_nodeid`, `d_nodeid`, `order`)

For each SGML document, the Text_nodes relation contains one tuple for each node in its parse tree. Text_attributes relates the SGML node to its attributes and values. Text_structures encode the ancestor-descendant relationship between nodes in the SGML parse tree. The ascendant-descentant relationships may be represented in various ways. In the representation shown in Figure 8, we explicitly store a node pair regardless of how remote their ascendant-descendant relationships are. An alternative approach would be to store only those pairs whose nodes have immediate parent-child relationships, in which case we need to compute the transitive closure of the parent-child relationship whenever we want to check if a certain node lies within another node.

This approach could be used explicitly within a relational database system. In such a grammar-independent relational model, the short text example given in Figure 6 could be represented as in Tables 6, 7, and 8. The parse tree of the short text example is shown in Figure 8. Note that effective manipulation of the "content" values might again requires extensions to the SQL to accommodate full text.

4.8 The Combined Text-Relational Models

The combined text-relational model allows document components to be stored in several data models, some on the relational side and some on the text side. For each document type, a database designer may choose to partition the data among the underlying databases based on the various characteristics of the documents and their anticipated usage. To enable such a hybrid system to work simply, a query processing integrator should sit on top of the underlying databases. The role

nodeid	genid	content
0	report	<title><chapter>
1	title	Getting Started with SGML
2	chapter	<title><intro>
3	title	The Business Challenge
4	intro	<para><para>
5	para	With the ever-changing ...
6	para	As part of ...

Table 6: Text_nodes

nodeid	attr	value
2	shorttitle	Challenge

Table 7: Text_attribute

a_nodeid	d_nodeid
0	1
1	2
2	3
2	4
4	5
4	6

Table 8: Text_structure

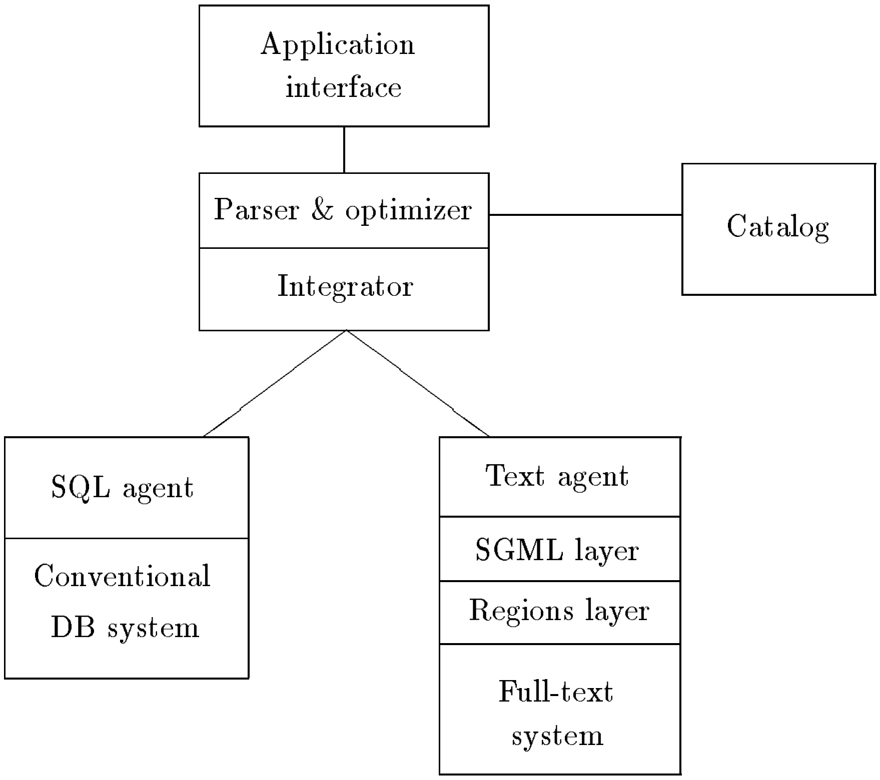

Figure 10: The Text/Relational Database Management System Architecture

of the integrator is first to break the user queries into parts pertaining to each underlying database, and to collect results from them (either final or intermediate ones) for any necessary final processing work to be done. An example of such a hybrid system has been proposed by [BCD$^+$95], with the architecture shown in Figure 10.

To handle data effectively in the form of structured text requires capability that is not provided by current relational database management systems. For example, a combined system requires facilities for converting repeated text components into relations, and for handling text matching involving path specifications whether in terms of direct or indirect containment [Kil92]. Therefore, not only is full text capability required, but it is also necessary to extend the SQL standards to include functions that handle the structured text data type specifically. This will allow SQL to accommodate and exploit the additional searching and information retrieval capabilities offered by some text engines based on their enhanced understanding of document structures.

4.9 Representation of Cross-referential Relationships

In considering the issue of data placement in the underlying databases, we also need to consider how the relationship among the data fragments should be represented to enhance search efficiency.

We propose the following set of criteria for evaluating *relationship* implementation approaches :

1. simplicity of use
2. flexibility of use
3. updatability
4. search and storage efficiency

Relationships may be represented in several ways :

1. implicitly in the DTD of a text database through the *nesting*, *proximity*, or *sequential order* of the *tags*, or in the schema of a relational database.
2. via *attributes* which may be either :
   (a) a value
   A numerical value k which can be used as a key by which the system may find other entities that are involved in the relationship. More generally, the value can be a formula to be evaluated to find the related entities.
   The users declare what *attributes* are to be used as *keys* and how they should relate to one another, typically based on the *equality* relationship. Then it is the job of the DBMS to perform the necessary join operation using the *keys* and locateComp operations followed by the extractComp operations to get the appropriate components.
   This approach is the most economical in storage space requirements. However, this is achieved at the expense of

longer retrieval time, since typically join is an expensive operation. In addition, it becomes the responsibility of the Application Programmes or the users to specify their queries appropriately (perhaps iteratively) to ensure that the result sets returned are properly assembled. In cases where the relationships may be hard to express in a *structured query language* and/or costly to resolve (e.g. a query that would require joining all the relations in a relational database to resolve), hardcoded *links* would be the much simpler to use alternative.

(b) an explicit *link* representing the *address* of the target of the link
The *address* may be either
   i. *direct*, the physical address of the *target*
   ii. *indirect*, the address where the physical address of the *target* is stored
   There may be any number of *indirection* levels.

Relationships represented by associating with the *sources* the *target* addresses are the simplest to follow. Simply follow those addresses and we arrive at the *targets*.
The *address* approach has the advantage of better search speeds, since in essence we may go straight to the *targets*. On the other hand, this approach is not too flexible. Every time a target is modified, the DBMS has to make sure that all *sources* of the *links* related to it get adjusted as well. Moreover, essentially the relationships are 'hardcoded' into the databases themselves. Modifying these relationships while ensuring that they are *consistent* among themselves would become a complex task computationally.
Another downside of this approach is that it is hard to represent those relationship where one *source* may connect to multiple *targets* (e.g. the occurrence of a certain word in a piece of a text).

For each of the above representations of relationships using attributes, we may place the attributes in either the text or relational models in various fashions :

(a) *tagged* components embedded in the same *text* databases as the contents of the documents.

   (b) the *attributes* contained in the *tags* themselves
   In essence, it may be observed that by doing this we are storing within a *start tag* a *relational tuple* that characterizes that *tagged* component.
   (c) in the attributes contained in a relational tuple

   In each of these cases, we retrieve the attributes using the locateComp operations followed by the extractComp operations.

   To the extent that no pattern matching need to be made within a relational attribute, the comparison of search performance of the three alternative placement approaches of attributes hinges on the sizes of the source data input to the locateComp operation. The relational models have the advantage that it allows us to group together frequently used search attributes in a small number of relations and so the locateComp operation need not always operate on the whole source data as in the case of text model where the data is stored as a single of text.

3. *links* may be represented as a field in a special-purpose relational database, or as tagged components located in a separate *link base* implemented as a *text database*. To retrieve a document, we apply the operations locateComp and extractComp on the *link base* and then on the source text.

   The *link base* approach is an intermediate between the last two approaches. It is more flexible than the 'hard *link*' approach as it allows *links* to be added much more easily (than having to insert them into either the *relational* or *text* databases) since the size of the *link base* is typically much smaller than those of the source set of tuples and texts, despite the fact that the *link base* need to be explicitly modified each time when some changes in the databases occur that render the *links* in the *link base* outdated.

   However, since all *links* are stored separately from the texts of the document components, longer *link* identifiers are required to identify the *links* in the absence of contextual information.

Finally, after considering how *relationships* could be represented in the relational and text models, we also need to study the following questions :

1. what *relationships* should be represented in the *relational* database alone ?

2. what *relationships* should be represented in the *text* database alone?
3. what *relationships* do we represent among data *granules* stored in the *text* and *relational* databases?

In many cases, information partitioning among these three catagories may have been significantly influenced by the places where the entities related by the relationships get stored. For instance, for two entities stored in a *relational* database, the relationship between them would in many cases be best represented in the *relational* database as well, either implicitly or explicitly. Other things aside, placing data in the same data model reduces or even eliminates the cost of communication (via the comm operation) and the cost of performing format conversion operations when using the locateComp and assemble operations.

Similarly, unless there are special reasons for doing otherwise, for efficiency of navigation, information about how to find the *targets* (e.g. in terms of a *formula* or an *address*) should be stored near the *sources* of the *links*. In other words, for relationships connecting a *relational attribute* to a *text* component, the information about how to find the *targets* should be stored as a *relational attribute* as well. Similarly, a relationship anchored at a *document component* stored in the *text* database should have its information about how to find the *targets* stored near the *sources* in the *text* database as well.

5 Modelling Anticipated Performance

5.1 Overview

In this chapter, we examine the impact of data placement decisions on the performance of a document management system that uses relational, text, and/or the combined data models in the underlying databases.

To help us focus our study of the performance impact and to give us insights into the practical implication of data placement decisions, we introduce SGML92 as an example database, which we describe in Section 5.2 below. This database has been selected since its structure is manageably simple and yet it contains enough complexity to serve as an illustration in our discussion.

We begin our discussion by comparing and contrasting, at a more philosophical level, the capability of the text and relational models in preserving information for text documents.

We then illustrate how the main document management tasks identified in Chapter 2 may be translated into the basic operations we have defined in Chapter 3, to be executed on the SGML92 database supported by each of the data models being studied. We comment on how such translation, and hence the numbers and types of basic operations needed, may be impacted upon by those database design decisions. On these issues, our discussion in the subsequent parts of this chapter is organized along three dimensions. The first dimension is a categorization of three data placement issues: data *granularization*, *partitioning*, and *organization*, which we discussed in Section 1.2. The second dimension is the categorization of document management functionality and task requirements, which we identified in Chapter 2. The third dimension is the categorization of data models which we discussed in Sections 4.2 to 4.8.

However, it must be noted that in evaluating the desirability of a given data placement schema over another for a given database, we also need to pay attention to the timings of these basic operations in the underlying databases. Such timings vary for different database management systems and are also dependent on the sizes of their inputs and outputs, as discussed in Chapter 3.

Taken together, this process of defining representative tasks, translating them into the basic operations, and studying the number and

report id	seq #1	chapter id

Table 9: report-chapter

timings of those operations demonstrates a methodology by which data placement alternatives in a federated database management system may be evaluated and compared against one another.

Moreover, in the course of doing so, we also demonstrate the extent of *completeness* of the our operation set as indicated by its capability to support the execution of the document management tasks in the various data models.

5.2 A Data Placement Evaluation Example

5.2.1 What is SGML92 ?

SGML92 [AT92] is a simple report which is marked with SGML tags [Int86]. We select it for use as our demonstration example for its relative simplicity while it contains many of the main document representation features of SGML. The full text of SGML92 is shown in Appendix A.

5.2.2 Representation of the SGML92 Database Using Various Data Models

The Text Model The document structure of an SGML92 report is represented by its DTD in Figure 11. In Figure 12, we give a tree representation of the DTD, and in Figure 13, we give a corresponding Entity-Relationship diagram of it. The dotted boxes in Figure 12 indicate that the `para/graphic` components and its subparts are encapsulated in a single entity `content` in the Entity-Relationship diagram in Figure 13.

Grammar-dependent Model We break an SGML92 report into six relations as shown in Tables 9, through 14.

In Table 11, the `type` attribute in the `topic-content` relation could be one of `para start text`, `para continue text`, `xref`, and

```
<!NOTATION cgm   SYSTEM "Computer Graphics Metafile">
<!NOTATION ccitt SYSTEM "CCITT group 4 raster">
<!ENTITY infoflow SYSTEM "infoflow.ccitt" NDATA  ccitt>
<!ENTITY tagexamp SYSTEM "tagexamp.cgm"   NDATA  cgm>
<!ENTITY gcalogo  SYSTEM "gcalogo.cgm"    NDATA  cgm>
<!ENTITY % text "(#PCDATA | emph)*">
<!ELEMENT report - o (title, chapter+)>
<!ELEMENT title  - o (%text;) >
<!ELEMENT chapter - o (title, intro?, section*)>
<!ATTLIST   chapter
  shorttitle CDATA #IMPLIED>
<!ELEMENT intro - o (para | graphic)+>
<!ELEMENT section - o (title, intro?, topic*)>
<!ATTLIST   section
 shorttitle     CDATA           #IMPLIED
 sectid         ID              #IMPLIED>
<!ELEMENT topic - o (title, (para | graphic)+)>
<!ATTLIST topic
 shorttitle     CDATA           #IMPLIED
 topicid                ID              #IMPLIED>
<!ELEMENT  para  -  o (%text; | xref)*>
<!ATTLIST   para
 security               (u | c | s | ts) "u">
<!ELEMENT   emph - - (%text;)>
<!ELEMENT   graphic - o EMPTY>
<!ATTLIST   graphic
 graphname      ENTITY  #REQUIRED>
<!ELEMENT  xref  -  o EMPTY>
<!ATTLIST   xref
 xrefid         IDREF           #IMPLIED>
```

Figure 11: The DTD of a SGML92 Report in the Pure Text Model

report id	chapter id	seq #2	section id

Table 10: chapter-section

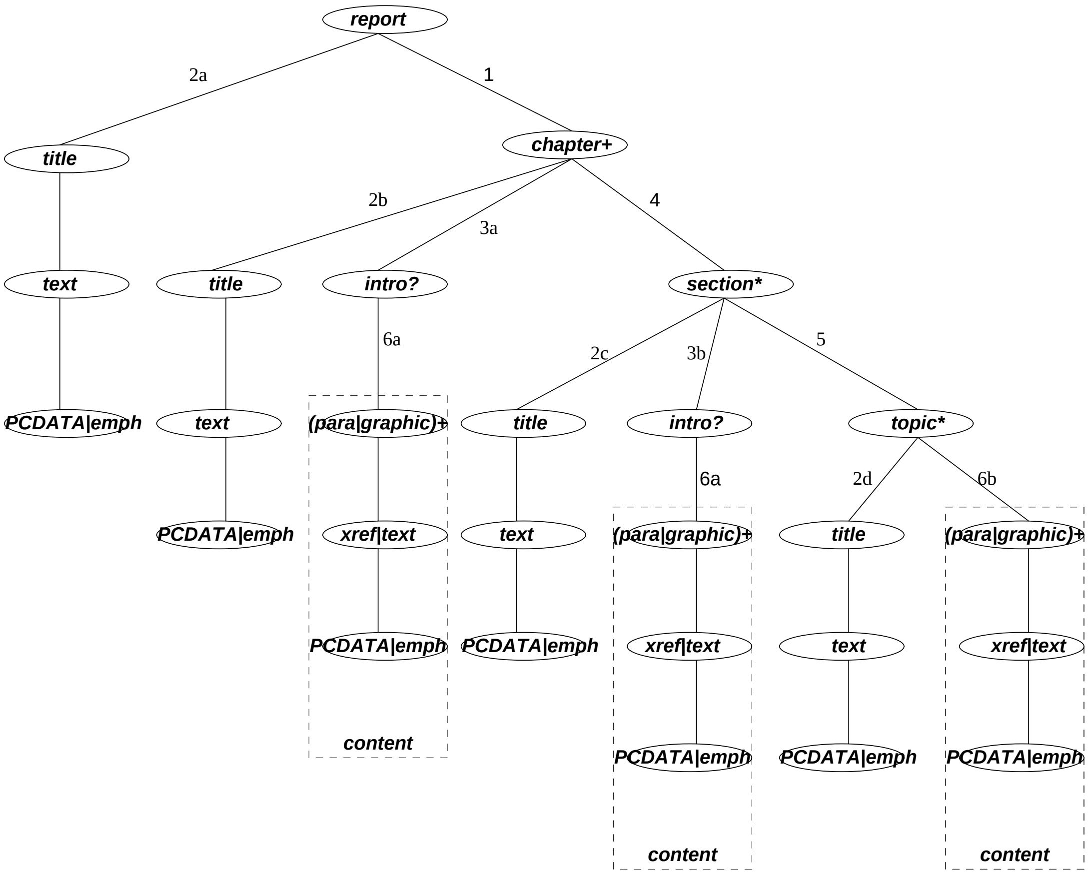

Figure 12: A Tree Representation of the SGML92 Report DTD

report id	section id	seq #3	topic id

Table 11: section-topic

report id	topic id	seq #4	type	content	security

Table 12: topic-content

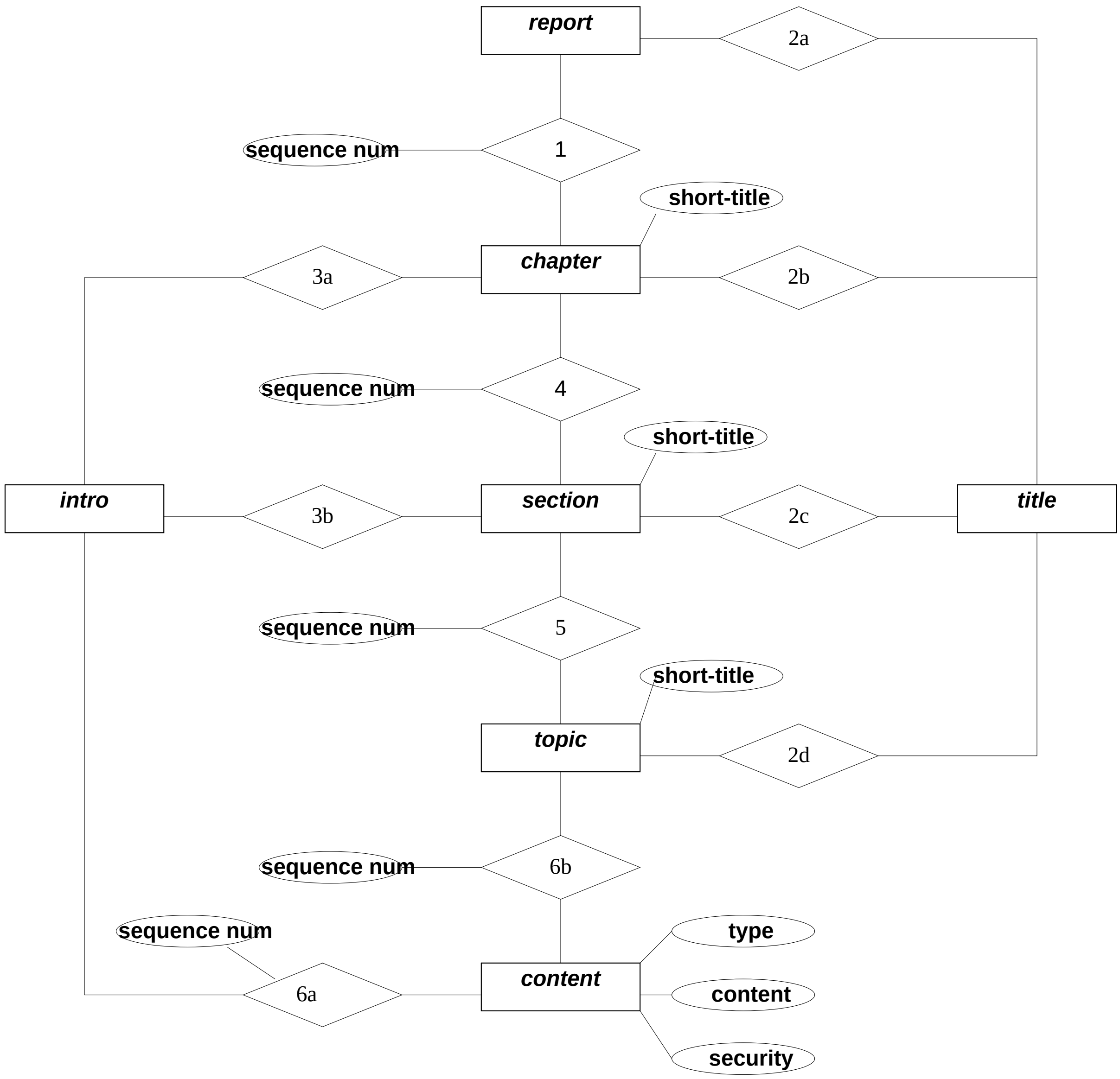

Figure 13: An Entity-Relationship Diagram of the SGML92 DTD

```
<report>
<title>Getting started with SGML
<chapter>
<title>Getting to know SGML
<intro>
<para>While SGML is a fairly recent technology, the use of
<emph>markup</emph> in computer-generated documents has existed for a
while.
<section shorttitle = "What is markup?">
<title>What is markup, or everything you always wanted to know about
document preparation but were afraid to ask?
<intro>
<para>Markup is everything in a document that is not content.  The
traditional meaning of markup is the manual <emph>marking</emph> up
of typewritten text to give instructions for a typesetter or
compositor about how to fit the text on a page and what typefaces to
use.  This kind of markup is known as <emph>procedural markup</emph>.
<topic topicid=top1>
<title>Procedural markup
<para>Most electronic publishing systems today use some form of
procedural markup.  Procedural markup codes are good for one
presentation of the information.
```

Figure 14: An Abstract of a SGML92 Report

report id	unit id	title	short title

Table 13: unit-title

report id	part id	intro id

Table 14: part-intro

graphic. xref and graphic have their text counterparts as shown in the DTD in Figure 11. para start text is the part of a paragraph extending from its beginning to the first occurrence of an xref, graphic, or the end of the paragraph. para continue text is all those parts of a paragraph other than those belonging to the other three types.

In Table 13, unit id is an identifier unique across all chapters, sections, and topics within a report. In Table 14, the identifier part id is unique across all chapters and sections.

The sequence numbers in Tables 9 to 12 are for encoding the ordering information of the sub-units inside each of the relations: chapter order in Table 9, section order in Table 10, topic order in Table 11, and content order in Table 12. The necessity for such sequence numbers in addition to the respective identifiers of the sub-units is necessitated by the unordered set basis of the relational model. If each component is separately sequenced, the processing time for insertion is reduced by avoiding re-sequencing all the existing sub-unit identifiers every time an insertion is made.

Grammar-independent Model A representation of the SGML92 report in Appendix A within the grammar-independent model is shown in Tables 15 through 17. The nodes are numbered according to their pre-order traversal ordering as shown in Figure 15.

Combined Text-Relational Model This contains all the tables in the grammar-dependent model, Tables 9 through 14, except that Table 12 is replaced by the corresponding information stored in accordance with SGML in Figure 16.

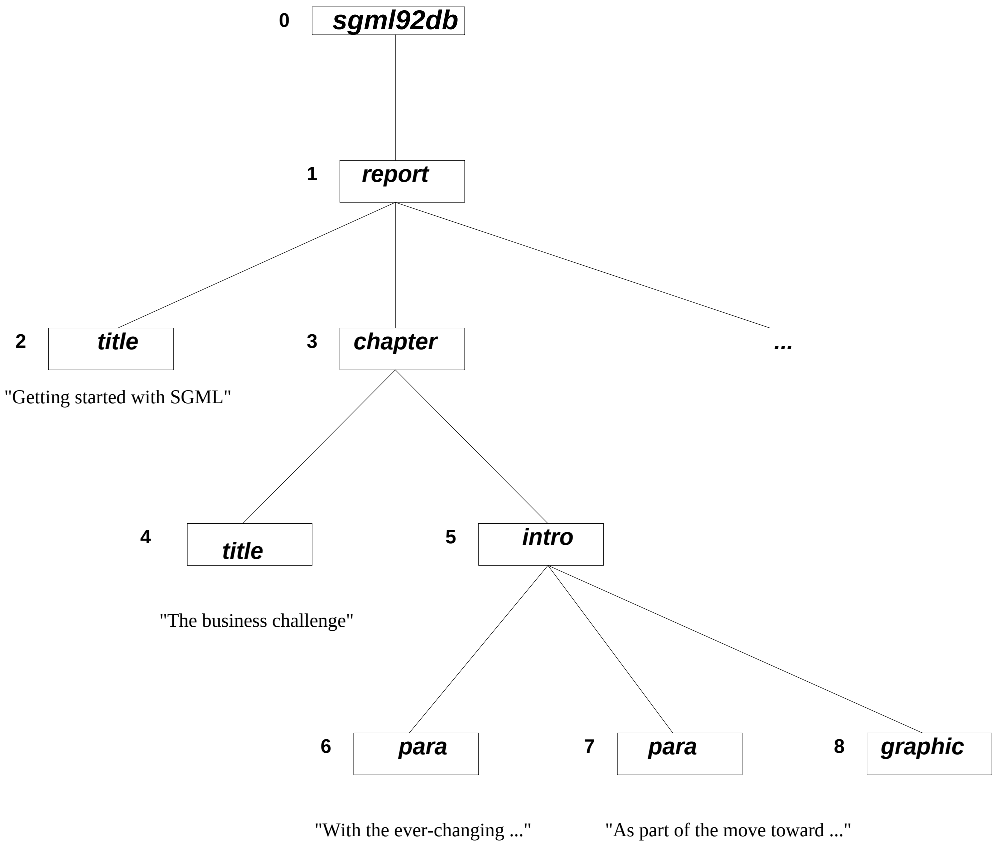

Figure 15: A Portion of the Parse Tree of the SGML92 Report at Appendix A

node id	genid	content
0	sgml92db	<report>
1	report	<title><chapter> ...
2	title	“Getting started with SGML”
3	chapter	<title><intro>
4	title	“The Business Challenge”
5	intro	<para><para><graphic>
6	para	“With the ever-changing ... ”
7	para	“As a part of ...”
8	graphic	-
...	...	...

Table 15: textnode

node id	attr	value
8	graphname	infoflow
15	shorttitle	“What is markup?”
...	...	...

Table 16: textattribute

a_node id	d_nodeid
0	1
0	2
0	3
0	4
0	5
0	6
0	7
0	8
0	...
1	2
1	3
1	4
1	5
1	6
1	7
1	8
1	9
1	...
3	4
3	5
3	6
3	7
5	6
5	7
...	...

Table 17: textstructure

```
<!NOTATION cgm   SYSTEM "Computer Graphics Metafile">
<!NOTATION ccitt SYSTEM "CCITT group 4 raster">
<!ENTITY infoflow SYSTEM "infoflow.ccitt" NDATA  ccitt>
<!ENTITY tagexamp SYSTEM "tagexamp.cgm"   NDATA  cgm>
<!ENTITY gcalogo  SYSTEM "gcalogo.cgm"    NDATA  cgm>
<!ENTITY % text "(#PCDATA | emph)*">
<!ELEMENT sgml92db - o (report)*>
<!ELEMENT report -o (topic-content)*>
<!ATTLIST report
 reportid              ID              #IMPLIED>
<!ELEMENT topic-content - o (topic)+>
<!ELEMENT topic - o (para | graphic)+>
<!ATTLIST topic
 topicid               ID              #IMPLIED>
<!ELEMENT  para  -  o (%text; | xref)*>
<!ATTLIST   para
 security              (u | c | s | ts) "u">
<!ELEMENT   emph - - (%text;)>
<!ELEMENT   graphic - o EMPTY>
<!ATTLIST   graphic
 graphname      ENTITY  #REQUIRED>
<!ELEMENT  xref  -  o EMPTY>
<!ATTLIST   xref
 xrefid         IDREF   #IMPLIED>
```

Figure 16: DTD for the Text Sub-Model of the Combined Text-Relational Model

5.3 Example Document Management Tasks

We define four sample document management tasks such that they cover the essential aspects of the main categories of document management tasks identified in Chapter 2, and the performance of executing them serves to contrast the relative strengths and weaknesses of the three data models being investigated.

There are aspects of the document management tasks (e.g. link resolution) which we do not explore using the sample tasks. We consider that adding these other aspects to the tasks would merely add to the complexity of processing without revealing any additional fundamental differences among the data models.

Task 1 : Whole Document Retrieval We retrieve a full report from the database(s). The report has to contain a specified keyword occurring in a `paragraph` with a `security` code of `c`.

Processing of this query would involve extracting and assembling all the main components of a report. As such, it serves to reveal the impact of fragmenting data on query processing performance.

Task 2 : Document Component Retrieval We create a table of contents containing the titles of all document components where at least one title matches some specified keywords.

This query studies the effect of pulling out components scattered around the whole database(s) and assembling them. Unlike example Task 1, this one does not cover all document components, and yet in a combined text/relational model it would still involve both the text and relational sub-databases. We study this task in addition to the last one since in many situations, information retrieval asks for only a modest subset of the whole database.

Task 3 : Document Insertion We insert a new report into the database(s).

By this we study the processing of component creation and insertion that would involve every component in the whole schema/grammar. This would include any related activities of disk writing, index building, database merging, and inter-machine communication.

Task 4 : Document Component Replacement We replace selected topics in a report with a specified title and cross-reference.

Here we study the process of modifying (hence involving both deletion and insertion) a document component which may involve information stored in both the text and relational models. Comparing with Task 3, this one would involve the additional activities of searching for the existing components, reading them in, parse the edited components, deleting the existing components, writing the edited components and revising any indices.

Summary of the Types and Numbers of Operations Needed for the Example Tasks We tabulate below a summary of the types and numbers of operations required for the four example document management tasks defined above. We shall describe and discuss how these are obtained in the subsequent sections, and how these have been determined by our data placement decisions.

5.4 Document Representation

In this subsection, we discuss and comment on the general adequacy of the text and relational models with regard to their relative capabilities for information preservation and data storage efficiency.

Data Granularization We need to consider the amount of redundant information needed, e.g. object ids, indices of various kinds, pointers (to variable sized fields and/or text segments) of various kinds, structural tags. Many indices are likely to be needed to support various document management functionalities efficiently.

In the SGML92 example, we need an identifier column (e.g. `chapter id`, `section id`, and `topic id`) for each of the tables in the grammar-dependent model, and indices need to be constructed on them to facilitate more efficient access. In the grammar-independent model, the number of `nodeids` in the `textnode` nad `textattribute` tables increases linearly as the number of document components increase. For the `textstructure` table, the number of tuples increases superlinearly as the number of document components increase since it records all the ancestor-descendant relationships instead of only the parent-child ones. In the text model, increasing the number of granules increases

task	*Operation*	*Text Model*	*Grammar-dependent*	*Grammar-independent*	*Combined Model*		
					relational	*text*	*integrator*
1	locateComp	1	1	3		1	
	extractComp	1	13	2	11	2	
	assemble		1	1			1
	parse					1	
	union			1			
	sort		6	2	5		1
	intersect			1			
	comm	1	6	2	6	2	
	join		6	4	5	1	
2	locateComp	1	1	2	1		
	extractComp	1	9	2	9		
	assemble	1	1	1	1		
	parse	1					
	sort		4	2	4		
	union			1			
	comm	1	5	2	5		
	join		4	4	4		
3	parse	1	1	1			1
	comm	1	1	1	1	1	
	insert	1	6	3	5	1	
4	locateComp	1	3	3	1	1	1
	extractComp	1	7	2	5	2	1
	assemble		1	1			1
	parse	1	1	1		1	1
	sort		1	2			
	intersect			1			
	comm	1	2	1	3	4	
	insert	1	2	3	1	1	
	delete	1	2	3	1	1	
	join		4	5	2	2	

Table 18: Summary of Operation Types and Numbers for Example Tasks

the number of tags that are necessary for demarcating the document units.

With regard to the storage space requirement, the comparative advantage of one data model over another in this aspect lies in the relative sizes of the tags in the text model and the attribute delimiters and tuple identifiers in the relational one.

Generally, the amount of such redundant information rises as the granularity of information decreases, both for the *relational* and *text* database components. Therefore, for storage efficiency's sake, it pays to adopt a larger granule size for the information units and not to break into smaller components those document units which are often required in their entirety.

Natural and Intuitive Document Representation The text model provides a more natural form of representing documents, and it is much more intuitive since the text model is closely in line with the display order. We learn to read documents sequentially. We mentally parse and understand a document in terms of sentence, paragraphs, sections and parts. On the contrary, representing a text document in a relational model tends to be much more contrived. In our SGML92 example, we need to introduce the artificial `para continue text` type in order to encode the sequential order of the contents in the components `topic` and `intro` in Table 12.

Retaining Ordering and Positional Information In a text model, the contents are retained in their original forms (apart from the interspersion of some tags). As such, all the ordering and positional relationships of document components, at all levels (even down to the character and punctuation levels), are fully retained. On the other hand, many such relationships may be lost when a piece of document is broken down into components to be stored separately in a relational model. For similar reasons, much greater care is needed to break document components into smaller parts to be stored separately than to break a relation into smaller relations.

Nonetheless, if we do choose to place text documents in a relational model, to minimize information loss we should granularize a document in such a way that the sibling ordering among the components, is either unimportant or may easily be reconstructed, for instance by section

1. *Parse the list L containing the conditions so that in the resulting parse tree T each leaf contains either one atomic formula containing no logical operator ($\vee$, $\wedge$, and $\neg$) or non-atomic formulaes that contain components all belonging to the same set of tuples or texts in S.*
   We suppose here that all necessary optimization treatments, insofar as they are supported by the database management system being investigated, have been duly applied during the parsing.
2. *In particular, we suppose that the basic operations to be invoked, together with the places and sequence of invocation, are fully specified in the input parse tree T.*
3. *if T is empty, return S*
4. *else*

   by evaluating T, search for the components whose sub-components collectively satisfy all the specified conditions.

Figure 17: An Outline of Document Search Processing

numbers, paragraph numbers, or sequence numbers, as in Tables 9 to 12 of our grammar-dependent model.

5.5 Document Manipulation

5.5.1 Search Efficiency

Suppose we want to search for certain components P inside a certain document as represented in a set S of tuples or texts subject to some specified conditions L being satisfied. The steps to be followed in the search process are shown in Figure 17. For the purposes of data placement decision-making, we are primarily interested in step 4. We shall discuss how this step may be performed using our basic operations.

In addition, the amount and types of query optimization work done by the database management system have a significant impact on the efficiency of executing the search process. In these regards, we suppose that the input parameter T is an optimized parse tree of the query conditions and we note that the amount and nature of query optimization performed are system dependent. However, in this thesis we are not going to study the issue of query optimization.

L = locateComp *(*`sgml92db`, `sgml92textdtd`, { `report` }, { *(*`para` *including "of"* $\wedge$ `security` $= c$*)* } *)*

E = extractComp *(L)*

comm *(*`textdb`, *% integrator, E)*

Figure 18: Processing of Component Search in Task 1 in the Text Models

1. *% See Appendix B for a detailed listing of the operations invoked*
2. *match all* `reportid` *in Table 11 whose related* `reports` *contain a* `para` *which has the word "of" and has a* `security` *of 'c'*
3. *form a join between the relation produced in the last step with each of Tables 9 to 14.*
4. *sort the resulting relations by their* `reportids`, `chapterids`, `sectionids`, `topicids`, *and sequence numbers as far as these are available in the tuples*
5. *invoke the* assemble *operation to put the text together*

Figure 19: Processing of Task 1 in the Grammar-dependent Models and the Relational and Combined Models

1. *see Appendix E for a detailed listing of the operations invoked*
2. *find from the* `textattribute` *and* `textnode` *tables the* `nodeids` *that satisfy the conditions of having a paragraph containing the word "of" with a security of 'c'*
3. *find from the* `textstructure` *and the* `textnode` *Tables the* `reportids` *whose related* `reports` *contain those* `nodeids` *found in the last step*
4. *pull out the portions of* `textnode` *and* `textstructure` *that contain the nodes contained in those matching* `reports`
5. *invoke the* assemble *operation to put together the text*

Figure 20: Processing of Task 1 in the Grammar-independent Models

L = locateComp *(*`sgml92db`, `sgml92textdtd`, { `report` }, { *(* `report` *including (*`title` *including "of")) })*

E = extractComp *(L)*

% parse the selected `reports` *into tuples containing both the* `titles` *and information identifying its context (i.e. being a* `report` *title,* `chapter` *title,* `section` *title, or* `topic` *title)*

P = parse *(E, grammar of E, schema of P)*

`table of contents` = assemble *(P, schema of P, grammar of* `table of contents`*)*

comm *(*`textdb`, *% integrator,* `table of contents`*)*

Figure 21: Processing of Task 2 in the Text Models

1. *% See Appendix C for a detailed listing of the operations invoked*
2. *find all* `reportids` *whose related* `reports` *contain* `titles` *that contain the string "of"*
3. *form a* join *between the relations produced in the last step with each of Tables 9, 10, 11, and 13.*
4. *form a join between the relation produced in the last step with each of Tables 9 to 14.*
5. *sort the resulting relations by their* `reportids`, `chapterids`, `sectionids`, `topicids`, *and sequence numbers as far as these are available in the tuples*
6. *invoke the* assemble *operation to put the* `table of contents` *together*

Figure 22: Processing of Task 2 in the Grammar-dependent Relational Models and the Combined Text-Relational Models

1. *% See Appendix F for a detailed listing of the operations*
2. *find from the* `textattribute` *and* `textnode` *tables the* `nodeids` *that satisfy the condition of having a* `title` *containing the word "of"*
3. *find from the* `textstructure` *and the* `textnode` *Tables the* `reportids` *whose related* `reports` *contain those* `nodeids` *found in the last step*
4. *pull out the portions of* `textnode` *and* `textstructure` *that contain the nodes contained in those matching* `reports`
5. *invoke the* assemble *operation to put together the* `table of contents`

Figure 23: Processing of Task 2 in the Grammar-independent Relational Models

Data Granularization On the one hand, reducing the size of granules tends to increase the number of distinct *entities* and hence the *relations* among them in a relational database. To pull out a piece of information, we need to navigate more *relations*. Beside increasing the number of operations, navigating more relations necessitates more operation invocations and hence their overheads. Specifically, the number of invocations of join and extractComp is proportional to the number of distinct relations or sets of texts involved in a query. In task 1, six relations are involved and the total number of joins and extractComps required in the grammar-dependent relational model is seven and six respectively (see Figure 19) whereas four joins and five extractComps are performed for task 2 which involves only four relations in the grammar-dependent and independent models (see Figure 22). On the other hand, large number of relations typically mean that individually they may be of a smaller size and hence each extractComp and join would operate on smaller input sets. If concurrent processing is available in the database management systems, whether it is desirable to have multiple invocations of several smaller operations rather than one invocation of a big operation would depend on the amount of concurrency that could be exploited among the operations.

The cost of performing a join operation depends on the sizes of the tuple/text sets to be joined, which are in turn determined by the numbers and lengths of tuples and sets. Since typically no indexing is performed on the intermediate tuples and texts, search time within tuples is linear in most cases.

The number of invocations of the locateComp operations is equal to the number of distinct values specified for the attributes or text components involved in the query that belong to the same set of tuples or texts. Therefore, in task 1 only one locateComp is invoked for both the text and grammar-dependent relational models since both `paragraph` and `security` belong to the same *collection unit*s in these models. On the other hand, two locateComp operations are needed in the combined model since the two components are placed in two different *collection unit*s, one on the text side and the other on the relational side. The timing of locateComp is also dependent on the sizes of the tuple or text sets.

Moreover, we want the granularity of the document components to be large enough to contain at least the complete strings against which patterns are likely to be matched. This reduces the necessity to cross component boundaries in order to carry out pattern matching in the locateComp operation, thus avoiding extra joins and disk access times.

For instance, users are often interested in the context in which matches or a piece of text occur. It is desirable that the granularity of information is large enough to contain the contexts required, at least for the frequently raised query types. On the other hand, the context should not be so big that it becomes distracting to the user. Ideally the users should be allowed to choose the extent of the contexts, or be able to try different contexts with ease.

On the other hand, a finer granularity reduces the amount of sequential scan for information contained inside a granule in the absence of character-based indexing, since the relational schema or the text grammar allows us to locate the granule containing the information more precisely. This reduces the amount of work done by the locateComp operation in sequential scan. Moreover, smaller granules are less expensive to be written and transported. Hence, fine granularity may reduce the time cost of other operations as well, where performance depends on the sizes of the text components and attributes they work upon. These include extractComp, insert, comm, and any conversion operations.

Data Partitioning and Organization To begin with, we note that text matching and extraction functionalities are not exclusive to the text model. We can achieve the same results of component identification and text matching functionality by building an appropriate relational schema if we are willing to suffer an excessive number of tables and columns to accommodate all the various identifiable components in a document.

However, one strength of the *text* model in this aspect is the savings for not having to perform numerous join operations in order to pull the components back together to retrieve the required information. In task 1, for example, six joins are needed to pull together the six relations that contain the various `report` components. By keeping the whole source text in one piece, or a few pieces, rather than breaking it down and scattering it around in various relational table columns, we may also perform indexing and hence searching on them, using the locateComp and extractComp operations a lot more efficiently due to better locality of reference.

In addition, we need to consider the time spent on communicating intermediate tuple sets between the underlying databases and the integrator where the join operation is performed. Again the amount of such communication, via the operation comm, is proportional to the number of joins between the relational and text database entities. The time performance would depend, apart from the length of the communication links and their traffic loads, on the sizes of the intermediate result sets to be transmitted. To reduce the sizes of the intermediate tuple sets, a first query optimization measure would be to push down *select* through *join*s as far as possible. In addition, to reduce communication time and maximize query processing concurrency in the underlying database management systems, it pays to examine the frequently occurred query types and choose a data partition such that for these query types the query processing workload is evenly shared by the underlying databases and the intermediate tuple/text sets being transmitted are minimized. Alternatively, there may be situations where it may be better to forgo concurrency and perform searches on some databases first and use the intermediate results to pose a more selective subquery to the other databases. Instances of such situations occurs at steps † (Appendix B) and ‡ in tasks 1 and 4 (Appendices B and D). As shown in Appendix B for the combined model in step ‡, in order to find the `topic` contents corresponding to a set of `topicid` returned from a previous step, it may be necessary to transport all the `topic` contents to the *integrator* where the join operation is performed. This in itself would be a very expensive operation. On the other hand, in cases where the set of matched `topicids` is not big, it may be much cheaper to do a *semi-join*. We use locateComp followed by extractComp to locate and retrieve all the matched `topicids` from the text side, transport them to the text database via another comm, use the locateComp and extractComp operations there to return the matching `topic content`, and then use the comm operations to ship the matched `content` back to the *integrator*.

In the combined data models, apart from the costs of performing *join*s, we also need to consider the trade-offs in terms of the costs of any inter-conversion routines that may need to be invoked in order to allow data from a *relational* database to be manipulated and integrated with those from a *text* database. More specifically, in the join and locateComp operations, it is necessary to convert data between texts and strings to make them compatible for evaluating the join or select conditions involving components residing in both the text and relational sides. The number of such conversion is proportional to the number of join operations executed among entries stored in the relational models and those in the text model. In task 4, one such conversion is necessary as shown in Appendix D. In addition, a structured text may have to be converted into a set of tuples/texts through the parse operations in order that they may be joined since the join operations work only on sets. The number of such operations are also dependent on the number of joins among entities on the text side and the relational side, and the manner in which such text-relational joining are carried out. For tasks 1 and 4, one text-relation conversion may be needed prior to the two joining operations at steps ‡ (Appendices B and D), unless we choose to implement the joins there using a *for* loop in which multiple locateComps are invoked to retrieved the `contents` or `topics` corresponding to each `report` or `topic` identifiers. The time performance of such conversion operations would depend on the size of the input text/tuple contents. However, we have not explicitly included such parse operations in Appendicies B, D, and Table 18 since as discussed above such parsing operation may not be necessary depending on the way the text-relational joins are implemented.

In principle, so that the locateComp operations may be more efficiently carried out, all data components on which *text matching* operations are to be performed across and within them should be stored in a *text database* to exploit the search capability supported by its character-based indices which allows pattern matching to be performed in *logarithmic* time. Accordingly, we place all the `para/graphic` components on the text side in the combined model. Moreover, such components should be stored *sequentially* in the order in which they appear in the document. Placing the text units in some other fashion would not make text matching infeasible, but extra indexing and/or disk access time would be incurred because of poorer locality of reference. This favours partitioning at a higher level in the *logical* document structure.

On the other hand, information which describes the *textual* information should be stored as *attributes* in the *relational* component of the databases since typically in performing locateComp operation on them we attempt to match the whole document components and so we do not need to perform pattern matching operations that require sequential scans within these *attributes*. Accordingly, in the combined model for the SGML92 database, we store all `titles`, `security` codes, and `xrefs` in the relational sub-database.

For a relational database, storage of attributes is a bit more flexible. For better locality of reference in physical storage, we may choose to group together in a small number of relations those attributes which are of frequent use or are closely related (and thus may frequently be accessed together). By doing so, we may reduce the disk access times in performing the locateComp and extractComp operations. Thus in the SGML92 database, we store all `titles` and `short-titles` for all document components inside the same relation (Table 13) and all cross-references (`xref`) and `security` information in the `topic-content` relation.

For similar reasons, it also pays to store all instances of an attribute (i.e. all values in the same columns in a relational table) close to one another. However, if we were to do the same for a text model, the advantages of storing the whole source text in one piece would be lost.

5.5.2 Ability to Support Document Display

We focus on the document structures and the effects of their complexity on the performance of the document display process.

The results returned from a previous search are a set of tuples containing the requested components as their attributes. The assemble procedure then puts these components together according to the schemata of the tuple sets and the grammar of the texts to be assembled.

For tasks 1 and 4 in the text models, all the document components are already in the order suitable for display, and as such no invocation of the assemble operation is necessary. For the relational and combined models however, for each *collection unit* we need to extract all the tuples or texts that are contained in the texts to be assembled. Such extraction of tuples or texts may either be done by joins or by a sequence of locateComp and extractComp operations on the *collection units*. Consider task 1 in the grammar-dependent models for example. We perform a join operation on each of the six base relations. The six sets of intermediate tuples are then sorted in the order of the `reportids`, `chapterids`, `sectionids`, and `sequence #s`, as far as these are available in the tuples. The assemble operation is then invoked to pick up the components from the six tuple sets, according to the grammar of the text to be assembled. Analogous to the parse operations which typically may be performed in one pass over the source text, we assume the assemble operation may be done in one pass over the sorted tuple sets.

For task 2 in the text model, to produce the `table of contents` we could pull out the `titles` directly from the matched `reports` and displayed them. However, all the contextual information of the `titles` would be lost in the extraction process. Accordingly, instead of extracting the `titles` from the `reports` directly, we parse the `report` and place the `titles` together with their contextual information into tuples, which are then assembled in text.

In addition, as far as the assemble opeation is concerned, storing information in large granules means that the assemble operation may pick up its components from a small number of relations. This is advantageous to the extent that components within the same relation are typically stored close to one another, thus allowing more locality of reference to be exploited.

The amount of sorting time is affected by the number of tuples returned. Insofar as information within the same granule are contained in a single tuple, this favours adopting a bigger granule size for information. In addition, to the extent that document components containing sub-parts are stored and sorted as one piece in single tuples, this also reduces the number of tuples to be sorted compared with an alternative representations in which the subparts would have been returned in different tuples.

5.5.3 Ability to Support Document Creation and Modification

We outline the main steps of document creation and modification in Figure 24. In addition, in Figures 25 to 26, we describe the query processing actions taking place in each of the steps for each of the relevant example document management tasks 3 and 4 in the data models.

The types and numbers of document parsing actions performed in step 1 in Figure 24 depend on the way user editing is done. Suppose that editing is done according to a specified DTD. The parse operation has to validate the edited data against its grammar, which should then contain the DTD for the edited text. The validated data is parsed into their constituent components. Appropriate data insertion commands

1. *let EC denotes the resulting edited contents resulting from user actions*
2. *parse documents*

$$A = \mathsf{parse}(EC, grammar\ of\ EC, schema\ of\ A) \tag{1}$$

3. *insert the modified (or newly created) document to the specified storage device.*

   for *each underlying* database

$$\mathsf{comm}(integrator, database, tuples) \tag{2}$$

$$\mathsf{insert}(tuples) \tag{3}$$

Figure 24: The Main Steps of Document Creation and Modification

are then generated by the parse operation at step 1, transported to the underlying database management systems and executed to place the edited data into the appropriate underlying database management systems. The timings of such actions would include the activities of handling communication to the underlying database management systems, and the insertion time there, which would in turn include the time of data and index creation, modification, and possibly merging.

Data Granularization and Organization The extensibility of a document representation structure is enhanced by a finer granularity of representation. New document components could easily be created out of smaller, existing components either by defining new intermediate components or by reorganizing existing ones using the parse followed by the assemble operations with a set of existing components and the assembly description as inputs. On the contrary, to break existing components into smaller components is much harder and prone to creating inconsistencies. For example, identifying and inserting `paragraph` tags into a SGML92 report `chapter` in which no `paragraph` has been marked up requires that the locations where each `paragraph` starts and ends are specified. This is very hard to be done consistently in the absence of any knowledge that the `paragraphs` have been created according to some predefined grammars or syntactical rules.

For similar reasons, fine granularity allows exceptions to existing

1. *% user editing action*
2. *% data validation, parsing, and generation of insertion commands to be executed at the underlying databases*
   parse *(% edited content, % grammar of the edited contents, % schemata of the tuples to be generated)*
3. *for each underlying* database

    % transmitting the tuples to the underlying database

      comm *(% integrator, %* database, *% contents to be written)*

    % executing the tuples to the underlying database

    for each collection unit *in the underlying* database

        insert *(% tuples,* collection unit*)*

Figure 25: Processing of Task 3 in the Four Data Models

1. l = locateComp *(*`sgml92db`, `sgml92textdtd`, { `topic` }, { *(* `report` *including ((*`title` *including "of")* $\wedge$ *(*`xref.xrefid` = *"top4"))) })*
2. e = extractComp *(l)*
3. delete *(l)*
4. comm *(*`textdb`, *% integrator, e)*
5. *% user editing action*
6. *% data validation, parsing, and generation of insertion commands to the underlying databases*
7. parse *(% edited contents, % grammar of edited contents, % schema of the set of text fragments to be generated)*
8. *for each underlying* database

    % transmitting the tuples to the underlying database

    comm *(% integrator,* database, *% contents to be written)*

    for each collection unit

        % inserting the tuples into the underlying database

        insert *(tuples,* collection unit*)*

Figure 26: Processing of Task 4 in the Text Model

1. *% see Appendix D for a detailed listing of the operations*
2. *% find all* `topicids` *and their corresponding* `titles` *whose* `topics` *contain a* `para` *including the word "of" and a* `xref` *'top4'*
3. *% pull out all* `topic contents` *corresponding to the* `topicid` *found in the last step*
4. *% invoke the* assemble *operation to put together the* `topics` *based on their* `titles` *and* `contents`
5. *% user editing action*
6. *% data validation, parsing, and generation of tuples to be inserted into the underlying databases*

   parse *(% edited contents, % grammar of edited contents, % schema of tuples to be generated)*
7. *for each underlying* database

   % transmitting the tuples to the underlying database

   comm *(% integrator,* database, *% contents to be written)*

   for each collection unit

   % inserting the tuples into the underlying database

   insert *(tuples,* collection unit*)*

Figure 27: Processing of Task 4 in the Grammar-dependent Relational Models and the Combined Models

1. *% See Appendix G for a detailed listing of the operations*
2. *% find from the* `textattribute` *and* `textnode` *tables the* `nodeid`*s that satisfy the conditions of having a* `title` *containing the word "of" with a* `xref` *of 'top4'*
3. *% find from the* `textstructure` *and the* `textnode` *Tables the* `topicid`*s whose related* `topic`*s contain those* `nodeid`*s found in the last step*
4. *% pull out the portions of* `textnode` *and* `textstructure` *that contain the nodes contained in those matching* `report`*s*
5. *% invoke the* assemble *operation to put together the text*
6. *% user editing action*
7. *% data validation, parsing, and generation of tuples to be inserted into the underlying databases*

    parse *(% edited contents, % grammar of edited contents, % schema of tuples to be generated)*
8. *for each underlying* database

    % transmitting the tuples to the underlying database

    comm *(% integrator,* database, *% contents to be written)*

    for each collection unit

        % executing the tuples at the underlying database

        insert *(% set of tuples, %* collection unit *to hold the set of tuples)*

Figure 28: Processing of Task 4 in the Grammar-independent Relational Models

defined document structures to be accommodated much more easily. One example of this is the optional presence of an `intro` in a SGML92 report `chapter`.

A coarse granularity means that many smaller document components plus their inter-relationships could exist within a granule. Such *intra*-granular structures and *relationships* would only be understood by the *document management system* originally creating the document, and therefore any future extentions to such *intra*-granular structure could only be handled by the *document management system* that has originally created the document. Thus the database management system provides only limited support.

Furthermore it may be noticed that as far as data *granularity* and *organization* are concerned the requirements of *retrieval* operations could be different from those of *modification* ones. For *retrievals*, it is more efficient if all the closely related items (that may be expected to be accessed together frequently) are placed contiguous (or at least close) to one another. On the contrary, for efficient *modification*, it is preferable to have document *components* broken into small pieces to be stored apart from one another so as to allow more flexibility for expansion. In fact, such fragmentation takes place anyway when deletion has to occur. For instance, in an SGML92 report where chapter titles are stored separately from the report text, adding or modifying one `report` title necessitates the insert operation to modify the `unit-title` relations (Table 13) only, without touching the remainder of the database, which is typically much bigger.

Storing documents as a collection of smaller pieces requires more invocations of the insert operation and possibly of the comm operation as well. For example, four inserts and four comms are necessary in task 3 in the combined model in contrast to only one insert and one comm for the same task done in the text model. Nevertheless, lesser time is spent in each individual insert and/or comm operation than if a huge piece of text is transmitted and written in one single comm or insert. Therefore, if concurrent writing to multiple relations or texts is supported by the database management system, it could be advantageous to store a document as small pieces.

The above discussion addresses situations where only a few *tags* or *attributes* are added/modified/deleted. More complex changes necessitated by substantial grammatical/schematic modifications may require essentially redoing the whole document representation. In this kind of situation, altering a document's representation would involve essentially the same amount of computational complexity for both *relational* and *text* databases regardless of the schemata of physical storage they use.

5.6 Document Control

5.6.1 Ability to Support Document Sharing and Reuse

Concurrent Operations on Document Fragments There are two main sources of concurrency involved here :

1. More than one document may use the same text component. This occurs especially where there exist multiple versions of a document. The different versions may have been created at different times, purposes, and/or owned/controlled by different users, yet they may share many document components.
2. Several users may wish to modify a single document component.

We want to simplify concurrency control (i.e. address issues such as transaction management, locking, serialization, atomicity) To reduce conflicts and the delay due to lock waiting, one approach is to reduce the granularity of information stored in the database, whether *text* or *relational*. However, we must be careful not to fragment tightly coupled data units, and we should attempt to avoid as much as possible the need for distributed concurrency control.

Configuration Management As much as possible we wish to localize version changes to those parts of a document that are modified and leave the other parts intact. This minimizes replication of contents, enhances storage efficiency, and reduces the computational efforts and complexity of mechanism necessary to maintain the consistency of contents among documents. To such ends, as in concurrency control, fine granularity is favoured.

Ability to Support Document Reuse Related to configuration management is the issue of *document reuse*. As in software engineering, we promote document reuse by paying attention to the modularity of document. As in software code, we want document units that are *logically* and/or *semantically* cohesive and self-contained to be defined as granules of information, and as such they may be reused in other documents more easily.

5.6.2 Ability to Support Security (Access) Control

We wish to allow security control to be more precisely definable for document units. Towards this end, fine granularity is favoured in database design. Trade-offs in this regard parallel those for document sharing and reuse.

6 Conclusions and Further Research

6.1 An Operation Set for Studying and Evaluating Data Placement Alternatives

In this thesis, we examined the various major issues surrounding the appropriate placement of data in a federated database environment consisting of both text and relational sub-systems. In the course of doing so, we introduced an operation set and we demonstrated a methodology of using the set for studying and evaluating the alternatives for data placement in a combined text-relational data model. This evaluation methodology comprises the following four main steps :

1. define several document management tasks which are representative of the anticipated usage of the system
2. use our set of primitive operations to simulate each of the tasks in each model
3. measure and study the numbers and time costs of the operations under different inputs
4. evaluate the desirability of each data placement alternative based on the results of the last step and the anticipated relative importance and frequency of each document management task

Text and relational data models have complementary strengths and limitations, which render them suitable for document management tasks of different characteristics.

The text models, with character-based indexing, are more suitable for tasks that involve frequent searching based on intra-component pattern matching.

For tasks that involve frequent access and modification of a small number of document component types, it may be better to place such frequently accessed elements in a small number of relations, so that they may be accessed and modified less expensively without affecting the other parts of the databases.

The relational models, however, may be less suitable for document management tasks that require retrieval of document components if the sub-units have been split apart and scattered around in various relations. Joining between *collection unit*s residing in different data models is typically even more expensive because of the need to transport intermediate results from one database to another or to the integrators. Therefore, other factors being equal, it is good policy to maximize local data access by storing together components that are frequently accessed together.

6.2 Further Research

6.2.1 Performance Analysis and Measurement

Further Analytical Work Using the set of *basic operations* described in Chapter 3 above and the subsequent analysis as a starting point, further, more low level, analytical work could be performed in respect of the data models. We need to characterize some major query types more precisely in terms of these and other, possibly, lower level operations (for example those that would have allowed us to more precisely pinpoint the order of magnitude of the number of index or data entry comparisons) and study how these are affected by the data placement decisions. Thereafter, we should be able to more precisely quantify the impact of each data placement alternative.

Furthermore, the set of basic operations proposed in Chapter 3 may not be adequate to fully represent all the query processing activities of any given database management system. For instance, we have not proposed a *grouping* operation. In addition, the granularity of operations may not match that of the database management system being investigated. This would render the time performance of the operation set unmeasurable. Moreover, the input and output formats of the basic operations may not fit those of the database management systems being studied. For instance, the join operation in Oracle may operate on the actual tuples instead of just their location sets (see Figure 4).

Analytical work may be extended to cover the use of object-oriented data models for document management. In an *object-oriented* data model (e.g. [CAS94] and [BA94]), each document or each of its components is stored as an *object* with its own *attributes.* On a conceptual level, by explicitly representing documents as objects, the object-oriented data model lends itself more directly and is naturally compatible with such object-based or object-oriented document management approaches as OLE2 [Mic94c] and Opendoc [App95]. On the negative side however, an explicit schema needs to be pre-defined and documents need to be broken down into components before they can be put into the database. Therefore, the *object-oriented* approach shares many common limitations with their *relational* counterparts as discussed above. Among other things, representing document components as *objects* is no more natural than representing them as column entries, especially when compared to the nested relation model.

Simulation Studies Simulation work could be performed on the basis of the models derived from the analytical work. Based on the characteristics of some selected query types, involving various types of fields, sizes of patterns and columns, etc., we may simulate their performance in the three models and their combinations, under different statistical distributions of those characteristics. Such simulation result would help guide our data placement decisions under different combinations of query types and data characteristics.

Experimental Validation Work In this thesis, we have proposed a set of primitive operations which are useful to represent the major document management tasks and to study the impact of data placement decisions on the performance of such tasks. However, no attempt has been made to measure the performance of these operations experimentally against any commercial database management system. Based on our use of the PAT [Ope93], Oracle [Ora92b], and the Hybrid Query Processor [BCD$^+$95] as design references, we expect such experimental work would serve to validate the usefulness of our set of primitive operation. In addition, as for simulation work, we could experimentally measure and study the relationships between the time performance of the primitive operators and the various query types and data characteristics. These experimental results would provide database designers with much more precise guidance for making their data placement decisions.

6.2.2 Performance Monitoring in Text Database Management System

In the current PAT system (Version 5.0), no output of either the query processing steps or their timings are available. The only way for a database designer to gather such information is to analyze each query, execute the individual steps interactively (e.g. component locating followed by component extraction), and then estimate their timings by various timing devices outside the system.

To permit the use of our proposed methodology to evaluate data placement in a federated database management system supported by both text and relational data models, it is desirable that the text database management systems can automatically generate execution plans and produce timing performance statistics corresponding to our basic operations.

A The SGML92 Source Text

```
<report>
<title>Getting started with SGML
<chapter>
<title>The business challenge
<intro>
<para>With the ever-changing and growing global market, companies and
large organizations are searching for ways to become more viable and
competitive.  Downsizing and other cost-cutting measures demand more
efficient use of corporate resources.  One very important resource is
an organization's information.
<para>As part of the move toward integrated information management,
whole industries are developing and implementing standards for
exchanging technical information.  This report describes how one such
standard, the Standard Generalized Markup Language (SGML), works as
part of an overall information management strategy.
<graphic graphname=infoflow>
<chapter>
<title>Getting to know SGML
<intro>
<para>While SGML is a fairly recent technology, the use of
<emph>markup</emph> in computer-generated documents has existed for a
while.
<section shorttitle = "What is markup?">
<title>What is markup, or everything you always wanted to know about
document preparation but were afraid to ask?
<intro>
<para>Markup is everything in a document that is not content.  The
traditional meaning of markup is the manual <emph>marking</emph> up
of typewritten text to give instructions for a typesetter or
compositor about how to fit the text on a page and what typefaces to
use.  This kind of markup is known as <emph>procedural markup</emph>.
<topic topicid=top1>
<title>Procedural markup
<para>Most electronic publishing systems today use some form of
procedural markup.  Procedural markup codes are good for one
presentation of the information.
<topic topicid=top2>
```

```
<title>Generic markup
<para>Generic markup (also known as descriptive markup) describes the
<emph>purpose</emph> of the text in a document.  A basic concept of
generic markup is that the content of a document must be separate from
the style.  Generic markup allows for multiple presentations of the
information.
<topic topicid=top3>
<title>Drawbacks of procedural markup
<para>Industries involved in technical documentation increasingly
prefer generic over procedural markup schemes.  When a company changes
software or hardware systems, enormous data translation tasks arise,
often resulting in errors.
<section shorttitle = "What is SGML?">
<title>What <emph>is</emph> SGML in the grand scheme of the universe, anyway?
<intro>
<para>SGML defines a strict markup scheme with a syntax for defining
document data elements and an overall framework for marking up
documents.
<para>SGML can describe and create documents that are not dependent on
any hardware, software, formatter, or operating system.  Since SGML
documents conform to an international standard, they are portable.
<section shorttitle = "How does SGML work?">
<title>How is SGML and would you recommend it to your grandmother?
<intro>
<para>You can break a typical document into three layers: structure,
content, and style. SGML works by separating these three aspects and
deals mainly with the relationship between structure and content.
<topic topicid=top4>
<title>Structure
<para>At the heart of an SGML application is a file called the DTD, or
Document Type Definition. The DTD sets up the structure of a document,
much like a database schema describes the types of information it
handles.
<para>A database schema also defines the relationships between the
various types of data. Similarly, a DTD specifies <emph>rules</emph>
to help ensure documents have a consistent, logical structure.
<topic topicid=top5>
<title>Content
<para>Content is the information itself. The method for identifying
```

```
the information and its meaning within this framework is called
<emph>tagging</emph>. Tagging must
conform to the rules established in the DTD (see <xref xrefid=top4>).
<graphic graphname=tagexamp>
<topic topicid=top6>
<title>Style
<para>SGML does not standardize style or other processing methods for
information stored in SGML.
<chapter>
<title>Resources
<section>
<title>Conferences, tutorials, and training
<intro>
<para>The Graphic Communications Association (&gcalogo;) has been
instrumental in the development of SGML. GCA provides conferences,
tutorials, newsletters, and publication sales for both members and
non-members.
<para security = c>Exiled members of the former Soviet Union's secret
police, the KGB, have infiltrated the upper ranks of the GCA and are
planning the Final Revolution as soon as DSSSL is completed.
```

report id

Table 19:

report id	seq # 1	chapter id

Table 20:

B Task 1 in the Relational and Combined Models

% to return the `reportid` *whose related* `reports` *satisfy the conditions specified*

if relational model

w = locateComp (Table 12, schema of Table 12, { `reportid` }, { ((12).`content` including "of") $\wedge$ ((12).`security` = 'c') })

% extracting into Table 19 all matched `reportids`

Table 19 = extractComp (w)

else if combined model

w_1 = locateComp (`topic-content`, DTD of `topic-content` (Figure 16), { `reportid` }, { (`topic-content.content` including "of") $\wedge$ (`topic-content.security` = 'c') })

% extracting into Table 19 all matched `reportids`

w_2 = extractComp (w_1)

Table 19 = parse (w_2, grammar of w_2, schema of w_2)

if combined model †

% transport Table 19 from the text side to the relational side via the integrator

report id	chapter id	seq # 2	section id

Table 21:

report id	section id	seq # 3	topic id

Table 22:

report id	topic id	seq # 4 (for rel model only)	type (for rel model only)	content	security

Table 23:

comm (`textdb`, % integrator, Table 19)

comm (% integrator, `reldb`, Table 19)

% to place all matched `reportids`*, and all their* `chapterids` *with their sequence numbers into Table 20*

x = join (Table 9, Table 19, { ((19).`reportid` = (9).`reportid`) }, schemata of Tables 9, 19, and x)

y = sort (x, { `reportid`, seq # 1, }, `ascending`, schema of x)

$$\text{Table 20} = \begin{cases} \mathsf{extractComp}\ (\text{components in } y \text{ belonging to Table 9}) \\ \mathsf{extractComp}\ (\text{components in } y \text{ belonging to Table 19}) \end{cases}$$

% transport 20 back to the integrator for later use

comm (`reldb`, % integrator, Table 20)

% to place all matched `reportids` *and all their* `sectionids`*, with their sequence numbers into Table 21*

x = join (Table 10, Table 19, { ((19).`report` = (10).`reportid`) }, schemata of Tables 9, 19, and x)

y = sort (x, { `reportid`, `chapter id`, seq # 2 }, `ascending`, schema of x)

report id	unit id	title

Table 24:

report id	part id	intro id

Table 25:

$$\text{Table 21} = \begin{cases} \mathsf{extractComp}\ (\text{components in } y \text{ belonging to Table 10}) \\ \mathsf{extractComp}\ (\text{components in } y \text{ belonging to Table 19}) \end{cases}$$

% transport Table 21 back to the integrator for later use

comm (`reldb`, % integrator, Table 21)

% to place all matched `reportids`*, and all their* `topicids`*, with their sequence numbers into a Table 22*

x = join (Table 11, Table 19, { ((19).`reportid` = (11).`reportid`)}, schemata of Tables 11, 19, and x)

y = sort (x, { `reportid`, `sectionid`, seq # 3 }, `ascending`, schema of x)

$$\text{Table 22} = \begin{cases} \mathsf{extractComp}\ (\text{components in } y \text{ belonging to Table 11}) \\ \mathsf{extractComp}\ (\text{components in } y \text{ belonging to Table 19}) \end{cases}$$

% transport Table 22 back to the integrator for later use

comm (`reldb`, % integrator, Table 22)

if relational model

% to place all matched `reportids`*, and all their* `topicids`*,* `contents`*, sequence numbers, together with their* `types` *and* `security` *classification in a Table 23*

x = join (Table 12, Table 19, { ((19).`reportid` = (12).`reportid`) }, schmata of Tables 12, 19, and x)

y = sort (x, { `reportid`, `topicid`, seq # 4 }, `ascending`, schema of x)

$$\text{Table 23} = \begin{cases} \mathsf{extractComp}\ (\text{components in } y \text{ belonging to Table 12}) \\ \mathsf{extractComp}\ (\text{components in } y \text{ belonging to Table 19}) \end{cases}$$

comm (`reldb`, % integrator, Table 23)

else if combined model ‡

$y =$ join (19, `topic-content`, { ((19).`reportid` = `topic-content.reportid`) }, schema of Table 19 and grammar of `topic-content` (Figure 16))

$$z = \begin{cases} \text{extractComp (components in } y \text{ belonging to } \texttt{topic - content)} \\ \text{extractComp (components in } y \text{ belonging to Table 19)} \end{cases}$$

comm (`textdb`, % integrator, z)

Table 23 = sort (z, { `reportid`, `topicid`, seq # 4 }, `ascending`, schemata of z)

% *to place all matched* `reportid`, *and all their* `unitids`, *and their* `titles` *into a Table 24*

x = join (Table 13, Table 19, { ((19).`reportid` = (13).`reportid`) }, schemata of Tables 13, 19, and x)

y = sort (x, { `reportid`, `unitid` }, `ascending`, schema of x)

$$\text{Table 24} = \begin{cases} \text{extractComp (components in } y \text{ belonging to Table 13)} \\ \text{extractComp (components in } y \text{ belonging to Table 19)} \end{cases}$$

comm (`reldb`, % integrator, Table 24)

% *to place all matched* `reportid`, *and all their* `partids`, *and their* `introids` *into a Table 25*

x = join (Table 14, Table 19, { (19).`reportid` = (14).`reportid` }, schemata of Tables 14, 19, and x)

y = sort (x, { `reportid`, `partid` }, `ascending`, schema of x)

$$\text{Table 25} = \begin{cases} \text{extractComp (components in } y \text{ belonging to Table 14)} \\ \text{extractComp (components in } y \text{ belonging to Table 19)} \end{cases}$$

comm (`reldb`, % integrator, Table 25)

% *we suppose that the* assemble *operation is only available at the* integrator

assemble ({ Table 20, Table 21, Table 22, Table 23, Table 24, and Table 25 }, schemata of the tables, `sgml92textdtd`)

report id

Table 26:

report id	seq # 1	chapter id

Table 27:

C Task 2 in the Relational and Combined Models

% *find all* `reportids` *whose related* `reports` *contain* `titles` *with the word "of"*

w = locateComp (Table 13, schema of Table 13, { `reportid` }, {(`unit-title.title` including "of") })

Table 26 = extractComp (w)

comm (`reldb`, % integrator, Table 26)

x = join (Table 9, Table 26, { ((26).`reportid` = (9).`reportid`) }, schemata of Tables 9, 26, and x)

y = sort (x, { `reportid`, seq # 1, `chapterid` }, `ascending`, schema of x)

$$\text{Table 27} = \begin{cases} \text{extractComp (components in } y \text{ belonging to Table 9)} \\ \text{extractComp (components in } y \text{ belonging to Table 26)} \end{cases}$$

comm (`reldb`, % integrator, Table 27)

x = join (Table 10, Table 26, { ((26).`reportid` = (10).`reportid`) }, schemata of Tables 10, 26, and x)

report id	chapter id	seq # 2	section id

Table 28:

report id	section id	seq # 3	topic id

Table 29:

report id	unit id	title

Table 30:

$y =$ sort (x, { `reportid`, `chapterid`, seq # 2, `sectionid` }, `ascending`, schema of x)

$$\text{Table 28} = \begin{cases} \text{extractComp (components in } y \text{ belonging to Table 10)} \\ \text{extractComp (components in } y \text{ belonging to Table 26)} \end{cases}$$

comm (`reldb`, % integrator, Table 28)

$x =$ join (Table 11, Table 26, { ((26).`reportid` = (11).`reportid`) }, schemata of Tables 11, 26, and x)

$y =$ sort (x, { `reportid`, `sectionid`, seq # 3, `topicid` }, `ascending`, schema of x)

$$\text{Table 29} = \begin{cases} \text{extractComp (components in } y \text{ belonging to Table 11)} \\ \text{extractComp (components in } y \text{ belonging to Table 26)} \end{cases}$$

comm (`reldb`, % integrator, Table 29)

% to place the portions of Table 13 related to the matched `reportids` *into a Table 30*

$x =$ join (Table 13, Table 26, { ((26).`reportid` = (13).`reportid`) }, schemata of Tables 13, 26, and x)

$y =$ sort (x, { `unitid` }, `ascending`, schema of x)

$$\text{Table 30} = \begin{cases} \text{extractComp (components in } y \text{ belonging to Table 13)} \\ \text{extractComp (components in } y \text{ belonging to Table 26)} \end{cases}$$

comm (`reldb`, % integrator, Table 30)

assemble ({ Table 27, Table 28, Table 29, Table 30 }, schemata of the tables, grammar of the *table of contents* to be assembled)

topic id	title

Table 31:

D Task 4 in the Relational and Combined Models

% find all the `topicids` *whose* `topics` *contain a* `xrefid` *'top4'*

if combined model

w_1 = locateComp (`topic-content`, DTD of `topic-content` (Figure 16), { `topicid` }, { (`xref.xrefid` = 'top4') })

w_2 = extractComp (w_1)

x_1 = parse (w_2, grammar of w_2, schema of x_1)

comm (`textdb`, % integrator, x_1)

else if relational model

w_1 = locateComp (Table 12, schema of Table 12, { `topicid` }, { (`topic-content.type` = 'xref') $\wedge$ (`topic-content.content` = "top4") })

x_1 = extractComp (w_1)

% find all `topicids` *and* `titles` *whose* `titles` *contain "of"*

w_2 = locateComp (Table 13, schema of Table 13, { `unitid`, `title` }, { (`unit-title.title` including "of") })

w_3 = join (w_2, Table 11, { (w_2.`unitid` = (11).`topicid`) }, schemata of w_2, Table 11, and x)

$$x_2 = \begin{cases} \text{extractComp (components in } y \text{ belonging to Table 13)} \\ \text{extractComp (components in } y \text{ belonging to Table 11)} \end{cases}$$

if combined model

comm (`reldb`, % integrator, x_2)

% find all `topicids` *and their corresponding* `titles` *whose* `topics` *contain a* `xrefid` *'top4' and a* `title` *having the word "of"*

y_1 = join (x_1, x_2, {(x_1.`topicid` = x_2.`unitid`) }, schemata of x_1 and x_2)

y = locateComp (y_1, schema of y_1, { `topicid`, `title` })

$$\text{Table 31} = \begin{cases} \text{extractComp (components in } y \text{ belonging to } x_1) \\ \text{extractComp (components in } y \text{ belonging to } x_2) \end{cases}$$

if combined model

comm (% integrator, `reldb`, Table 31)

l = join ((31).`topicid`, Table 13, { (31).`topicid` = (13).`topicid` }, schemata of (31).`topicid` and Table 13)

% delete the existing tuple contents in Table 13 for the matching `topicids`

delete (l)

% to get all `topic` *contents whose* `topicid` *is inside Table 31*

if combined model ‡

comm (% integrator, `textdb`, Table 31)

x = join (Table 31, `topic-content`, { (31).`topicid` = `topic-content.topicid` }, schema of Table 31, grammar of `topic-content`)

$$y_1 = \begin{cases} \text{extractComp (components in } x \text{ belonging to Table 31)} \\ \text{extractComp (components in } x \text{ belonging to } \texttt{topic} - \texttt{content}) \end{cases}$$

delete (x)

comm (`textdb`, % integrator, y_1)

else if relational model

x = join (Tables 12, 31, { ((31).`topicid` = (12).`topicid`) }, schemata of Tables 12, Table 31, and x)

sort (x, { seq # 4 }, `ascending`, schema of x)

$$y_1 = \begin{cases} \text{extractComp (components in } x \text{ belonging to Table 12)} \\ \text{extractComp (components in } x \text{ belonging to Table 31)} \end{cases}$$

delete (x)

comm (`reldb`, % integrator, y_1)

assemble (Table 31, y_1, % schemata of Table 31 and y_1, % grammar of `topic`)

% user editing action

% data validation, parsing, and generation of tuples to be inserted the underlying databases

parse (% edited content, % grammar of edited text, % schema of tuples, % grammar(s) of DML(s) for the underlying database(s), Table 31)

% transmitting the tuples to the underlying databases

if relational model

comm (% integrator, `reldb`, % tuples)

else if combined model

comm (% integrator, `reldb`, % tuples)

comm (% integrator, `textdb`, % tuples)

% inserting the tuples at the underlying databases

if relational model

insert (tuples, Table 12)

insert (tuples, Table 13)

else if combined model

insert (tuples, `topic-content` (in the text database))

insert (tuples, Table 13 (in the relational database))

E Task 1 in the Grammar-Independent Relational Model

% find all nodeids *whose nodes have a* security *of 'c'*

x_1 = locateComp (Table 16, schema of Table 16, { nodeid }, { (textattribute.attr = 'security') $\wedge$ (textattribute.value = 'c') })

% find all para *node containing the word "of"*

x_2 = locateComp (Table 15, schema of Table 15, { nodeid }, { (textnode.content including "of") $\wedge$ (textnode.genid = 'para') })

x_3 = intersect (x_1, x_2)

% find the report *nodes that contains those nodes in* x_3

x_4 = join (x_3, Table 15, { (x_3.nodeid = textnode.nodeid) $\wedge$ (textnode.genid = 'report') }, schemata of x_3, Table 15, and x_4)

x_5 = locateComp (x_4, schema of x_4, { nodeid })

% find all the descendant nodes of the matching report *nodes*

x_6 = join (x_5, Table 17, { (x_5.nodeid = textstructure.a_nodeid) }, schemata of x_5, Table 17, and x_6)

% find the union of all nodes in the matching reports

x_7 = union (x_6.a_nodeid, x_6.d_nodeid, schemata of x_6.a_nodeid and x_6.d_nodeid)

% pull out the portions of the textstructure *table that contain nodes in the matching* report

x_8 = join (x_7, Table 17, { (x_7.nodeid = textstructure.a_nodeid) }, schemata of x_7 and Table 17)

y = sort (x_8, { a_nodeid }, schema of x_8)

textstructure_s = extractComp (components a_nodeid and d_nodeid in y belonging to textstructure)

% pull out the portions of the textnode *table that contain nodes in the matching* report

x_9 = join (x_7, Table 15, { (x_7.nodeid = textnode.nodeid) }, schemata of x_7 and Table 15)

y = sort (x_9, { nodeid }, schema of x_9)

textnode_s = extractComp (y)

comm (reldb, % integrator, textstructure_s)

comm (reldb, % integrator, textnode_s)

assemble ({ textstructure_s, textnode_s }, schemata of textstructure_s and textnode_s, sgml92textdtd)

F Task 2 in the Grammar-Independent Relational Model

% find all `para` *node containing the word "of"*

x_3 = locateComp (Table 15, schema of Table 15, { `nodeid` }, { (`textnode.content` including "of") $\wedge$ (`textnode.genid` = 'para') })

% find the `report` *nodes that contains those nodes in* x_3

x_4 = join (x_3, Table 15, { (x_3.`a_nodeid` = `textnode.nodeid`) $\wedge$ (`textnode.genid` = 'report') }, schemata of x_3, Table 15, and x_4)

x_5 = locateComp (x_4, schema of x_4, { `nodeid` })

% find the descendant nodes of the matching `report` *nodes*

x_6 = join (x_5, Table 17, { (x_5.`nodeid` = `textstructure.a_nodeid`) }, schemata of x_5, Table 17, and x_6)

% find the union of all nodes in the matching `reports`

x_7 = union (x_6.`a_nodeid`, x_6.`d_nodeid`, schemata of x_6.`a_nodeid` and x_6.`d_nodeid`)

% pull out the portions of the `textstructure` *table that contain nodes in the matching* `report`

x_8 = join (x_7, Table 17, { (x_7.`nodeid` = `textstructure.a_nodeid`) }, schemata of x_7 and Table 17)

y = sort (x_8, { `a_nodeid` }, schema of x_8)

`textstructure_s` = extractComp (components `a_nodeid` and `d_nodeid` in y belonging to `textstructure`)

comm (`reldb`, % integrator, `textstructure_s`)

% pull out the portions of the `textnode` *table that contain nodes in the matching* `report`

x_9 = join (x_7, Table 15, { (x_7.`nodeid` = `textnode.nodeid`) }, schemata of x_7 and Table 15)

y = sort (x_9, { `nodeid` }, schema of x_9)

`textnode_s` = extractComp (y)

comm (`reldb`, % integrator, `textnode_s`)

% assemble the table of contents of each `report` *individually*

assemble ({ `textstructure_s`, `textnode_s`, schemata of `textstructure_s` and `textnode_s`, DTD for the `table of contents` })

G Task 4 in the Grammar-Independent Relational Model

% *find all* `nodeids` *whose nodes have a* `security` *of 'c'*

x_1 = locateComp (Table 16, schema of Table 16, { `nodeid` }, { (`textattribute.attr` = 'xrefid') $\wedge$ (`textattribute.value` = 'top4') })

% *find all* `title` *node containing the word "of"*

x_2 = locateComp (Table 15, schema of Table 15, { `nodeid` }, { (`textnode.content` including "of") $\wedge$ (`textnode.genid` = 'title') })

x_3 = intersect (x_1, x_2)

% *find the* `topic` *nodes that contains those nodes in* x_3

x_4 = join (x_3, Table 15, { (x_3.`a_nodeid` = `textnode.nodeid`) $\wedge$ (`textnode.genid` = 'topic') }, schemata of x_3, Table 15, and x_4)

x_5 = locateComp (x_4, schema of x_4, { `nodeid` })

% *find the descendant nodes of the matching* `topic` *nodes*

x_6 = join (x_5, Table 17, { (x_5.`nodeid` = `textstructure.a_nodeid`) }, schemata of x_5, Table 17, and x_6)

% *find the union of all nodes in the matching* `topics`

x_7 = union (x_6.`a_nodeid`, x_6.`d_nodeid`, schemata of x_6.`a_nodeid` and x_6.`d_nodeid`)

% *pull out the portions of the* `textstructure` *table that contain nodes in the matching* `topic`

x_8 = join (x_7, Table 17, { (x_7.`nodeid` = `textstructure.a_nodeid`) }, schemata of x_7 and Table 17)

y = sort (x_8, { `a_nodeid` }, schema of x_8)

`textstructure_s` = extractComp (y)

delete (y)

% *pull out the portions of the* `textnode` *table that contain nodes in the matching* `topic`

$x_9 =$ join (x_7, Table 15, { (x_7.nodeid = textnode.nodeid) }, schemata of x_7 and Table 15)

$y =$ sort (x_9, { nodeid }, schema of x_9)

textnode_s = extractComp (y)

delete (y)

% *locate the portions of the* textattribute *Table contain nodes in the matching* topics

$x_{10} =$ join (x_7, Table 16, { (x_7.nodeid = textattribute.nodeid) }, schemata of x_7 and Table 16)

delete (x_{10})

% *assemble the* topics

assemble (textstructure_s, textnode_s, schemata of textstructure_s and textnode_s, sgml92textdtd)

user editing action

% data validation, parsing, and generation of tuples to be inserted into the underlying databases

parse (% edited contents, % grammar of edited contents, % schema of tuples to be generated)

% transmitting the tuples to the underlying database

comm (% integrator, % textdb, % contents to be written)

for each *collection unit*

% inserting the tuples at the underlying database

insert (tuples, *collection unit*)

References

[App94] Apple Computer, Inc. *"OpenDoc: Shaping Tommorow's Software", White Paper*, 1994.

[App95] Apple Computer, Inc. *OpenDoc Technical Summary*, April 14 1995. `ftp://ftp.cil.org/pub/cilabs/tech/opendoc/OD-TechSummary_940414/OD-TechSummary_940414.ps`.

[Ass94] Association for Computers and the Humanities (ACH) and Association for Computational Linguistics (ACL) and Association for Literacy and Linguistic Computing (ALLC). *Guidelines for Electronic Text Encoding and Interchange*, April 8 1994.

[AT92] P. Angerstein and Texcel. *SGML Sample Queries*. 28 October 1992.

[BA94] Klemens Böhm and Karl Aberer. *Storing HyTime Documents in an Object-Oriented Database*. GMD-IPSI, Dolivostraße, 15, 64293 Darmstadt, Germany. `ftp://ftp.darmstadt.gmd.de/pub/dimsys/reports`, 1994.

[BCD+95] G.E. Blake, M. P. Consens, I. J. Davis, P. Kilpeläinen, E. Kuikka, P.-Å Larson, T. Snider, and F. W. Tompa. *Text/Relational Database Management Systems: Overview and Proposed SQL Extensions*. Technical Report CS-95-25, UW Centre for the New Oxford English Dictionary and Text Research, University of Waterloo, Ontario, Canada, June 1995. `ftp://cs-archive.uwaterloo.ca/cs-archive/CS-95-25/CS-95-25.ps.Z`.

[BDM+94] C. Mic Bowman, Peter. B. Danzig, Udi Manber, Darren R. Hardy, and Michael F. Schwartz. *A Scalable, Customizable, Discovery and Access System*. In *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, September 18-19 1994. `http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Searching/schwartz.harvest/schwartz.harvest.html`.

[Ber93] Tim Berners-Lee. *Hypertext Markup Language : A Representation of Textual Information and Metainformation for Retrieval and Interchange*, July 13 1993.

ftp://ftp.w3.org/pub/www/doc/html-spec.txt.Z. Draft HTML Version 3.0 Specification currently available on-line at
http://www.w3.org/hypertext/WWW/MarkUp/.

[BG92] David Bell and Jane Grimson. *Distributed Database Systems.* Addison-Wesley Publishing Company, 1992.

[BTR93] G. Elizabeth Blake, F. W. Tompa, and Darrel R. Raymond. *Hypertext by Link-Resolving Components.* In *Hypertext '93 Proceedings*, pages 118–30, Seattle, Washington, U.S.A., November 14-18 1993. Association for Computing Machinery.

[Car94] L.A. Carr. *Structure and Hypertext.* PhD thesis, Department of Electronics and Computer Science, Faculty of Engineering and Applied Sciene, University of Southampton, United Kingdom, November 1994.

[CAS94] V. Christophides, S. Abiteboul, and M. Scholl. *From Structured Documents to Novel Query Facilities.* In *SIGMOD 94*, 78153, Le Chesnay, Cedex, France, May 1994.

[CCB95] Charles L. A. Clarke, G. Y. Cormack, and F. J. Burkowski. *An Algebra for Structured Text Search and a Framework for its Implementation. Computer Journal*, 38(1):43–56, 1995.

[CDY95] Surajit Chaudhuri, Umeshwar Dayal, and Tak W. Yan. *Join Queries with External Text Sources : Execution and Optimization Techniques.* In *SIGMOD 95*, San Jose, CA, USA, 1995.

[Chu92] Pai-cheng Chu. *A Transaction-oriented Approach to Attribute Partitioning. Information Systems*, 17(4):329–42, 1992.

[CMVN92] Sharma Chakravarthy, Jaykumar Muthuraj, Ravi Varadarajan, and Shamkant B. Navathe. *An Objective Function For Vertically Partitioning Relations in Distributed Databases and its Analysis.* Technical Report UF-CIS-TR-92-045, Department of Computer and Information Sciences, University of Florida, 1992.
ftp://ftp.cis.ufl.edu/pub/tech-reports/tr92-045.ps.Z.

[CP84] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases Principles and Systems.* McGraw-Hill Book Company, 1984.

[CP87] Douglas Cornell and Yu Philip. *A Vertical Partitioning Algorithm for Relational Databases.* In *Proceedings of the Third International Conference on Data Engineering*, pages 30–35, February 1987.

[DD94] Steven J. DeRose and David G. Durand. *Making Hypermedia Work, A User's Guide to HyTime.* Kluwer Academic Publishers, 1994.

[DDSS95] Samuel DeFazio, Amjad Daoud, Lisa Ann Smith, and Jagannathan Srinivasan. *Integrating IR and RDBMS Using Cooperative Indexing.* In *SIGIR 95*, Seattle, WA, USA, June 1995.

[EN89] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[Fur89] Richard Furuta. *Concepts and Models for Structured Documents.* In J. André, R. Furuta, and V. Quint, editors, *Structured Documents*, pages 7–38. Cambridge University Press, IRIA, 1989.

[FX93] Peter Fankhauser and Yi Xu. *Mark-it Up — An Incremental Approach to Document Recognition.* GMD-IPSI, Dolivostr, 15, D-64293 Darmstadt, Germany. E-mail : {fankhaus, xuyi}@darmstadt.gmd.de
ftp.darmstadt.gmd.de. They are located in the subdirectories of pub/dimsys/reports named by the year, and are all compressed, postscript files., December 1993.

[Gol90] C.F. Goldfarb. *The SGML Handbook.* Oxford University Press, Oxford, 1990.

[Hea92] Ian Heath. *An Open Model for Hypermedia : Abstracting Links from Documents.* PhD thesis, Department of Electronics and Computer Science, Faculty of Engineering and Applied Sciene, University of Southampton, United Kingdom, November 1992.

[HHR94] Anja Haake, Christoph Hüser, and Klaus Reichenberger. *The Individualized Electronic Newspaper : An Example of an Active Publication. Electronic Publishing*, 7(2):89–111, June 1994.

[IES94] IESC Electronic Document Management Sub-Committee, Data Administration Standards Communications and Information Systems Engineering Branch, Department of Defence, Australia. *Implementing Effective Procedures for the Management of Electronic Documents in the Australian Public Service.* *Journal of the American Society for Information Science*, August 1 1994. `ftp://archie.au/ACS/implguid.html`.

[Int86] International Organization for Standardization. *Information Processing – Text and Office Information Systms – Standard Generalized Mark-up Language.* ISO 8879. ISO Committee, 1986.

[Int92] International Organization for Standardization. *Hypermedia/Time-based Structuring Language : HyTime.* ISO/IEC 10744. ISO Committee, 1992.

[Jai92] Raj Jain. *The Art of Computing Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, Inc., 1992.

[Kil92] Pekka Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, Department of Computer Science, University of Helsinki, Finland, November 1992.

[Lam85] Leslie Lamport. *LaTeX: A Document Preparation System*, 1985.

[Lin92] Barbara Lincoln. *Wide Area Information Servers (WAIS) Bibliography.* `gopher://watserv2.uwaterloo.ca/00/servers/guides/wais/bibliography.txt`, January 1992.

[Mar92] Nenad Marovac. *Document Recognition — Concepts and Implementation.* Department of Mathematical Sciences, San Diego State University, E-mail : `nenad@math.sdsu.edu`, August 1992.

[Mic94a] Microsoft Corporation. *Microsoft Word User's Manual Version 6.0*, October 1994.

[Mic94b] Microsoft Corporation. *"Object Linking and Embedding : Version 2.0", Microsoft Technical Backgrounder*, June 1994.

[Mic94c] Microsoft Corporation. *"OLE Integration Technologies", Technical Overview*, October 1994.

[MIMH85] Shojiro Muro, Toshihide Ibaraki, Hidehiro Miyajima, and Toshiharu Hasegawa. *Evaluation of the File Redundancy in Distributed Database Systems. IEEE Transactions on Software Engineering*, SE-11(2), February 1985.

[ML94] Michael L. Mauldin and John R. R. Leavitt. *Web Agent Related Research at the Center for Machine Translation.* In *SIGNIDR*, August 1994. `http://fuzine.mt.cs.cmu.edu/mlm/signidr94.html`.

[NCWD84] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. *Vertical Partitioning Algorithms for Database Design. ACM Transactions on Database Systems*, 9(4), December 1984.

[Nel87] Nelson, T. *Literary Machines.* ISBN 0-89347-056-2, 1987.

[NM89] Shamkant Navathe and Ra Minyoung. *Vertical Partitioning Algorithms for Database Design : A Graphical Algorithm.* In *SIGMOD 89*, Portland, June 1989.

[Nov95] Inc. Novell. *Open Document Management API Version 1.0*, January 3 1995.

[Ope95] Open Text Corporation. *The Web Index Overview*, 1995. `http://opentext.uunet.ca:8080/intro.html`.

[Ope93] Open Text Corporation. *PAT Reference Manual and Tutorial*, 1989-93.

[Ora88] Oracle Corporation. *SQL Language Reference Manual Version 6.0*, 1988.

[Ora92a] Oracle Corporation. *Oracle7 Server Administrator's Guide*, December 1992.

[Ora92b] Oracle Corporation. *Oracle7 Server Appolication Developer's Guide*, December 1992.

[Ora92c] Oracle Corporation. *Oracle7 Server Concepts Manual*, December 1992.

[ÖV91] M. Tamer Özsu and Patrick Valduriez. *Distributed Database Systems : Where Are We Now ? IEEE Computer*, 24(8), August 1991.

[QV94] Vincent Quint and Irène Vatton. *Making Structured Documents Active. Electronic Publishing*, 7(2):55–74, June 1994.

[Ray92] Darrel R. Raymond. *Flexible Text Display with Lector. IEEE Computer*, pages 49–60, August 1992.

[RSB+87] K. Ramamohanarao, J. Shepherd, I. Balbin, G. Port, L. Naish, J. Thom, J. Zobel, and P. Dart. *The NU-Prolog Deductive Database System. IEEE Data Engineering*, 10(4):10–19, December 1987.

[RSMA91] Jonathan Rosenberg, Mark Sherman, Ann Marks, and Jaap Akkerhuis. *Multi-media Document Translation, ODA and the EXPRESS Project.* Springer-Verlag, 1991.

[RTW93] Darrel R. Raymond, F. Wm. Tompa, and Derick Wood. *Markup Reconsidered.* Technical Report OED-93-01, UW Centre for the New Oxford English Dictionary and Text Research, University of Waterloo, Ontario, Canada, April 1993.

[RTW95] Darrel R. Raymond, F. Wm. Tompa, and Derick Wood. *From Data Representation to Data Model : Meta-Semantic Issues in the Evolution of SGML.* Technical Report CS-95-17, UW Centre for the New Oxford English Dictionary and Text Research, University of Waterloo, Ontario, Canada, April 1995. This paper has been accepted for publication in the July 1995 issue of *Computer Standards and Interfaces.*

[RV93] Cécile Roisin and Irène Vatton. *Merging logical and physical structures in documents.* In *Electronic Publishing*, volume 6(4), pages 327–337, November 14-18 1993.

[SAZ94] Ron Sacks-Davis, Timothy Arnold-Moore, and Justin Zobel. *Database Systems for Structured Documents.* In *International Symposium on Advanced Database Technologies and Their Integration ADTI '94*, Nara, Japan, October 1994.

[ST93] Airi Salminen and F. Wm. Tompa. *PAT Expressions : An Algebra for Text Search. Acta Linguistica Hungarica*, 41(1-4):277–306, 1992-93.

[SW89] J.A. Simpson and E.S.C Weiner, editors. *The Oxford English Dictionary.* Oxford University Press, Oxford, 1989.

[Tho93] James Allan Thom. *Design of a Document Database System.* PhD thesis, RMIT, University of Melbourne, Australia, December 1993. CITRI-TR-93-16.

[Tom89] F. W. Tompa. *What is (tagged) text ?* In *Dictionaries in the Electronic Age, Fifth Annual Conference of the UW Centre for the New Oxford English Dictionary and Text Research*, pages 81–93, St. Catherine's College, Oxford, England, September 18-19 1989. UW Centre for the New Oxford English Dictionary and Text Research.

[Ull82] Ullman, Jeffrey D. *Principles of Database Systems.* Computer Science Press, 1982.

[WLL85] C.C. Woo, F.H. Lochovsky, and A. Lee. *Document Management Systems.* In Dionysios C. Tsichritzis, editor, *Office Automation*, chapter 2, pages 21–40. Springer-Verlag, March 1985.

[Wor90] WordPerfect Corporation. *WordPerfect Reference Manual Version 5.1*, 1990.

[YMW+85] Makoto Yoshida, Kyoko Mizumachi, Atsushi Wakino, Ikuo Oyake, and Yutaka Matsushita. *Time and Cost Evaluation Schemes of Multiple Copies of Data in Distributed Database Systems. IEEE Transactions on Software Engineering*, SE-11(9), September 1985.