Design of Document Database Systems

Thesis submitted for the degree of
Doctor of Philosophy

by

James Allan Thom

Department of Computer Science
The University of Melbourne

1993
Abstract

This thesis seeks to provide a framework for the design of document database systems. Within this framework we address issues of physical and logical design of document databases and document database systems. These issues include the choice of data model, the query language requirements, physical schema design, logical schema design, and an underlying model of text.

Features and inadequacies of some existing data models for representing document databases are identified. We describe two data models based on complex objects. One model is a record-based model used by the Atlas database system, which has been developed at RMIT and The University of Melbourne. We use this model to describe physical storage structures. The other model is a higher-level object-based model which can be used to describe the logical structure of the database as seen by users, application developers, or database administrators.

A text query language (TQL) is described that is based on nested relations, has special text operators, and allows access to complex objects via reference attributes. TQL queries can be directly made on both logical and physical data models for complex objects. We describe a mapping into our logical model from entity-relationship diagrams, and a mapping into our physical data model. The mapping of queries from the logical to the physical level is also addressed.

A new approach to logical schema design using complex objects is presented; a resulting logical schema allows easy querying of structured documents and can be mapped into any one of several equivalent nested relational physical schemas.

In order to compare the efficiency of different structures for document databases it is desirable to have a reasonably accurate model of document text. A common
model of the distribution of words in text is the Poisson approximation to the binomial distribution. A shortcoming of the Poisson model is that it ignores the effect of clustering — the property that if a word occurs at least once in a document then it is likely to occur again. Our analysis of several document collections shows that the Poisson approximation can significantly overestimate the probability that a document contains a word. We propose a new model that describes the distribution of words in text, and show how this model can be used to estimate the probability that a document contains a word and estimate the number of distinct words in a document.

Using our text clustering model, we analyse the efficiency of different physical designs for document databases; we are particularly interested in the effect of partitioning large documents into smaller fragments. Our results suggest that for fast retrieval it is usually more efficient to partition large documents.
Preface

I would like to thank my supervisor Professor Kotagiri Ramamohanarao for stimulating my interest in database research. The thesis is based in part on the following research papers of which I have been a co-author [53, 69, 93, 117, 118, 136, 138, 139, 140, 141, 147, 157, 158], and I would like to thank all those with whom I have had the pleasure to work collaboratively including Mr Michael Fuller, Dr Alan Kent, Mr Eric Mackie, Dr Lee Naish, Professor Ron Sacks-Davis, Professor Kotagiri Ramamohanarao, Mr Peter Wallis, Dr Ross Wilkinson, and Dr Justin Zobel.

Parts of this thesis have been substantially based on previously published papers. Chapter 1 includes, in Section 1.3, material from “An architecture for hyperbase systems” which was presented at the First Australian Multi-Media Communications and Applications and Technology Workshop [158]. Chapter 2 includes, in Section 2.5, includes material from the paper “Schema design for complex objects” presented at Third Australian Database Conference [139]; this paper is also the basis for Chapter 4. Chapter 3 is based on the paper “TQL: a nested relational query language” that appeared in the Australian Computer Journal [136]. Chapter 3 also includes, in Section 3.3.7, material from the paper “Atlas: A nested relational database system for text applications” [117] that is to appear in IEEE Transactions on Data and Knowledge Engineering; this paper is also the basis for Section 6.1 of Chapter 6. Chapter 5 is based on the paper “A model for word clustering” that appeared in the Journal of the American Society for Information Science [141]. Chapter 6 is based on the paper “Efficiency of nested relational document database systems” which was presented at the Seventeenth International Conference on Very Large Data Bases [157].
The remainder of the thesis is entirely my own work except where due acknowledgement has been made in the body of the thesis. However, I would particularly like to thank Professor Kotagiri Ramamohanarao, Dr Justin Zobel and Mrs Margaret Thom who spent many hours reading this thesis and providing invaluable feedback. I would also like to thank all the others who have commented on parts of this thesis or the papers that proceeded it, including Professor Ron Sacks-Davis, Ms Leanne Salau, Dr Peter Smith, Dr Ross Wilkinson, and numerous anonymous referees.

During my PhD candidature I have been an employee, first in the Department of Computer Science at The University of Melbourne, and later in the Department of Computer Science at Science at the Royal Melbourne Institute of Technology. I would like to thank my superiors in both departments for allowing me time to work towards this thesis. While at these institutions the research groups in which I have worked have been supported by various research grants, including the Machine Intelligence Project (the Australian Government Department of Science), the Australian Research Council, the Artificial Intelligence Research Scheme, a Generic Industry Research and Development grant (Australian Government Department of Industry, Technology and Commerce), and had the support of the Collaborative Information Technology Research Institute, the Key Centre for Teaching and Research in Knowledge Base Systems, and the Centre for Intelligent Decision Systems.

A special thanks to Justin who did much of programming associated with Chapters 5 and 6, and to Alan who did the implementation of Atlas and the graph+ package used in preparation of this thesis.

Finally, and most importantly, I would like to thank my wife Leanne, daughter Katrina, and now son Nicholas, who have had to put up with me during difficult times throughout the research and writing of this thesis.

This thesis contains less than one hundred thousand words.

James Thom
March 1993
# Contents

1 Introduction .............................................. 1
   1.1 What is a document database system? ................. 2
   1.2 Characteristics of document databases .................. 4
      1.2.1 Nature of text ................................... 5
      1.2.2 Document Structure ................................. 7
      1.2.3 Querying a document database ...................... 8
   1.3 Hyperbase systems ..................................... 11
      1.3.1 An architecture for hyperbase systems ............ 14
      1.3.2 Nodal schema ...................................... 14
      1.3.3 Interface layer .................................... 15
      1.3.4 Node layer ........................................ 17
      1.3.5 Database layer .................................... 18
   1.4 Overview of thesis .................................... 19

2 Data Models ............................................... 21
   2.1 Data models and database schemas ...................... 22
   2.2 Semantic data models .................................. 25
      2.2.1 Entity-relationship model ......................... 25
      2.2.2 Other semantic data models ....................... 31
   2.3 Classical data models .................................. 32
      2.3.1 Hierarchical data model ........................... 33
      2.3.2 Network data model ................................. 35
      2.3.3 Relational data model .............................. 37
2.4 Extensions to classical data models .................................. 41
  2.4.1 Nested relational data model .................................. 41
  2.4.2 Nested sequences of tuples data model ......................... 44
  2.4.3 Deductive data model .......................................... 47
  2.4.4 Object-oriented data models .................................. 48
2.5 Complex object data models ........................................... 49
  2.5.1 Physical object data model of Atlas ............................. 49
  2.5.2 Logical data model with complex objects ...................... 54
2.6 Summary .................................................................. 57

3 Query Languages ................................................................ 58
  3.1 Formal nested languages ............................................. 62
    3.1.1 Nested calculus .................................................. 64
    3.1.2 Nested algebras .................................................. 66
    3.1.3 Nested sequences of tuples algebra .......................... 68
  3.2 SQL-based languages .................................................. 69
    3.2.1 SQL/NF ......................................................... 69
    3.2.2 NF2 ................................................................ 71
    3.2.3 SQL/W ......................................................... 74
  3.3 TQL ................................................................. 76
    3.3.1 Select–from–where ............................................. 76
    3.3.2 Explicit and implicit joins .................................... 78
    3.3.3 Support for text attributes ................................... 80
    3.3.4 Table and tuple operators ................................... 82
    3.3.5 TQL data manipulation language ......................... 86
    3.3.6 TQL data definition language ............................... 87
    3.3.7 Atlas architecture for TQL .................................. 90
    3.3.8 Limitations of TQL .......................................... 92
  3.4 Summary .................................................................. 93
# List of Figures

1.1 Examples of markup ........................................... 6  
1.2 Logical and layout structures in ODA .......................... 9  
1.3 Layers of a hyperbase system architecture .................. 14  
1.4 Graphical representation of a hyperbase node .............. 16  

2.1 Evolution of data models .................................... 25  
2.2 An E-R diagram for a hyperbase .............................. 27  
2.3 ISA relationships ............................................. 30  
2.4 A hierarchical schema for a hyperbase ........................ 34  
2.5 A Bachman diagram for the network representation of the hyperbase ....... 36  
2.6 A relational schema for a hyperbase .......................... 39  
2.7 A nested relational schema for a hyperbase ................. 43  
2.8 An NST schema for a hyperbase .............................. 46  
2.9 An SQL3 schema for a hyperbase ............................. 50  
2.10 An Atlas schema for a hyperbase ........................................... 52  
2.11 Database schema based on complex objects ............... 55  

3.1 A tuple in an instance of the node relation .................. 59  
3.2 Three tuples in an instance of the document relation ........ 60  
3.3 Declaration in SQL/NF for hyperbase ......................... 70  
3.4 Declaration in NF² for hypertext database .................. 73  
3.5 Declaration in SQL/W for hyperbase .......................... 75  
3.6 Atlas system architecture diagram ............................ 91  

4.1 Relational database schema ..................................... 100
4.2 Database schema nested on documents ........................................ 102
4.3 Database schema nested on authors ........................................... 103
4.4 Database schema nested on nodes ............................................. 104
4.5 Database schema nested on links ............................................. 105
4.6 Database schema with simple acyclic references ......................... 111
4.7 Minimal database schema with references ................................ 112
4.8 Database schema with duplication of references ......................... 113
4.9 Virtual database schema implied by references .......................... 114
4.10 Logical database schema based on complex objects ..................... 118
4.11 Ammended E-R diagram ....................................................... 121
4.12 Database schema with separate table for associations ................. 123
4.13 Logical database schema based on physical schema .................... 126

5.1 Relationship between $p$ and $p_n$ on Comact ............................. 134
5.2 Observed, Poisson-estimated, and cluster-estimated $p_n$ for $n = 100$
on Comact ................................................................. 137
5.3 Observed, Poisson-estimated, and cluster-estimated $p_n$ for $n = 1000$
on Comact ................................................................. 138
5.4 Observed, Poisson-estimated, and cluster-estimated $p_n$ for $n = 10000$
on Comact ................................................................. 139
5.5 $k = \frac{-\log_e(1-p_n)}{p}$ against $p_n$ for $n = 1000$ on Comact ........ 144
5.6 Median $\tilde{k} = \frac{-\log_e(1-p_n)}{p}$ against $n$ on Comact ............. 145
5.7 Distribution of $p_n$ values for $p = 10^{-6}$ and $n = 1000$ on Comact .. 149
5.8 Distribution of $p_n$ values for $p = 10^{-5}$ and $n = 1000$ on Comact .. 150
5.9 Number of distinct terms in each document of Comact ................ 152

6.1 A generalized signature file architecture .................................... 160
6.2 A schema for monolithic storage of documents ........................... 163
6.3 A schema for segmented storage of documents ........................... 164
6.4 A schema for duplex storage of documents ................................ 165
6.5 Predicted sizes of indexes and databases against fragment size ....... 170
6.6 Single-word queries by title against fragment size and against word probability ........................................ 175
6.7 Single-word queries by fragment content against fragment size for $p(w) = 10^{-7}$ ........................................ 178
6.8 Single-word queries by fragment content against fragment size for $p(w) = 10^{-6}$ ........................................ 179
6.9 Single-word queries by fragment content against fragment size for $p(w) = 10^{-5}$ ........................................ 180
6.10 Two-word queries by fragment content against fragment size for both $p(w) = 10^{-4}$ ........................................ 181
6.11 Three-word queries by fragment content against fragment size for each $p(w) = 10^{-4}$ ........................................ 182
6.12 Three-word queries by document content against fragment size for each $p(w) = 10^{-4}$ ........................................ 185
List of Tables

5.1 Size of each document collection ..................................... 132
5.2 Number of fragments in each document collection ............... 133
5.3 $\alpha$ and $\beta$ values for each document collection .............. 146
5.4 Typical $p, p_n, pois_n$, and $clus_n$ values on Comact ........... 147

6.1 Estimated number of distinct words $W(n)$ in fragment of size $n$ . . . 167
Chapter 1

Introduction

People have been recording information since before the dawn of history; the earliest known “document databases” were made by cave dwellers as they recorded events and stories on cave walls. The development of written language (“text”) began a process whereby an increasing volume of written information was recorded and stored, and the invention of the printing press accelerated the growth in the collection of written information. Early computers were able to manipulate symbolic information such as text but because of the limited size of computer memory only a very small amount of information could be stored. Enormous increases in the capacity of computer memory, both main memory and secondary storage, now permit storage of very large collections of documents. There is a very rapidly expanding market for document management systems, the size of the market in the United States has been forecast to double in 1993 to nearly four billion US dollars.¹ Effective management of large and small collections of electronic documents requires well designed document database systems.

In this thesis we seek to provide a framework for the design of document database systems. Within this framework we examine choices for logical design, and for physical design, of document databases. How can we determine the best designs?

Document databases lack the kind of structure which is typical of conventional database systems. A relational database system can easily store text from a highly

¹Sun presentation in 1992 quoting IDC source.
structured source such as an administrative form; for example, in a data entry system attributes in the database correspond to fields of a form displayed on a screen. Nevertheless, conventional database systems are poorly equipped to handle unformatted (free) text — text data which contains no uniform structure.

In this introductory chapter we begin by defining our terminology in Section 1.1, introducing concepts such as document, and document database system. The characteristics of document databases are considered in Section 1.2 together with the types of query access that may be required.

As an example of a document database system, we describe a possible architecture for a hypertext database system (or hyperbase system) in Section 1.3. The architecture has three layers, the bottom layer being a database system. In subsequent chapters we describe and analyse possible database schemas for this example using a variety of data models. In the final section, we give an overview of the issues considered in this thesis concerning the design of document database systems.

1.1 What is a document database system?

We begin by defining a component of a document.

**document fragment**

A document fragment is an excerpt of contiguous items such as a passage of text or a picture image.

Document fragments are excerpts from complete works called documents. Individual documents can be hierarchically structured; many documents contain several sections, each of which may contain several subsections or other logical units of text such as paragraphs or tables.
document

A document is a hierarchically structured collection of document fragments, representing a complete work.

Examples of a document are an article in an academic journal, an Act of Parliament, and a transcript of a speech. In some cases, the choice of exactly what constitutes a “complete work” can be arbitrary. Do we consider the entire Bible as a single document or each book of the Bible as a separate document?

In this thesis, we are primarily concerned with documents containing only text, and we are particularly interested in fragments in such documents.

text fragment

A text fragment is a document fragment containing only text.

text document

A text document is a document containing only text fragments.

Since most of the documents considered in this thesis contain only text, we often use the term document to mean text document.

We want to be able to collect similar documents in one place, one way to ensure some degree of similarity is to only include documents from a single source.

document collection

A document collection is a set of documents from a single source.

Even so, in one document collection, such as the Commonwealth Acts of Australia, the structure of individual documents may vary considerably. Nevertheless, in storing documents in a database system, we may wish to include documents from several different sources.

document database

A document database is a database of documents from one or more document collections.

Although a document database may contain documents from more than one document collection, these are usually on related topics. Since the documents in a
document database are entire texts, a document database is sometimes described as a *full text* database.

Conventional database systems are designed to support the storage and retrieval of data with simple, repetitive structure. Database systems for storing documents have particular requirements that distinguish them from conventional database systems.

**document database system**

A *document database system* is a database system that has been designed to support document databases.

Document database systems require specialized data structures, indexes, and interfaces. A document database system should support hierarchical data structures, so that documents can be treated both as single undivided objects and as complex objects containing many smaller parts. In a document database system it should be possible to retrieve documents on the basis of content. Achievement of efficient retrieval on document content requires indexing that provides access to individual parts of documents as well as to whole documents. Data retrieved from a document database system should be displayed with care as users cannot effectively view entire documents on a single screen; the system should support display of document fragments, and include a facility for document browsing.

### 1.2 Characteristics of document databases

The design of document databases and document database systems depend upon the principal characteristics of text, documents, and queries. In this section we discuss the nature of text, the structure of documents, and identify the types of queries that need to be supported in document databases. As part of this discussion we briefly review two related international standards: the Standard Generalized Markup Language (SGML) [63], a language for describing the structure of text, and the Office Document Architecture (ODA) [64], a model for describing the structure and layout of documents.
1.2.1 Nature of text

Text can be considered as an interleaved sequence of words and non-words. We need to choose which strings of characters constitute words and which do not. For example, ‘robot’ is clearly a word but what about ‘R2D2’? In our analysis of text in Chapter 5 we assume that a word is a series of alphabetic characters flanked by non-alphabetic characters. The exact definition of word depends on the application; in some cases it may be more appropriate to assume a word is a sequence of alphanumeric characters.

Documents may also contain character strings which do not constitute part of the text as such but indicate how the text ought to be displayed; these strings are called markup. Different text systems use different methods of markup; Coombs et al. [35] have categorized four general types of markup, examples of which are shown in Figure 1.1.

punctuational markup

Punctuational markup is the addition of punctuation, such as spaces between words, periods to end sentences, uppercase letters for proper nouns, and so on.

presentational markup

Presentational markup is given by the layout and appearance of a document, such as font changes and indentation.

procedural markup

Procedural markup is the embedding of commands within a document to describe formatting such as font changes and indentation.

descriptive markup

Descriptive markup is the embedding of commands within a document to describe its structure but not how it is to be displayed.

The variations between these methods of markup present a problem in building a general document database system that handles documents from different sources.
CHAPTER 1 — INTRODUCTION

No Markup
shorttitle1thisactmaybecitedasthecommonwealthofaustraliaconstitutionact

Punctuational Markup
(e.g. plain typewritten text)

Short title
1. This Act may be cited as the Commonwealth of Australia Constitution Act.

Presentational Markup
(e.g. Wordstar, MacWrite, …)

Short title
1. This Act may be cited as the Commonwealth of Australia Constitution Act.

Procedural Markup
(e.g. Ditroff, \TeX, …)

\verb+
+ This Act may be cited as the Commonwealth of Australia Constitution Act.

Descriptive Markup
(e.g. SGML, some of \LaTeX)

<sec>
<st>Short title</st>
<sn>1.</sn>
<p>This Act may be cited as the Commonwealth of Australia Constitution Act.</p>
</sec>

Figure 1.1: Examples of markup (based on a similar figure by Coombs et al. [35])
One solution is to use the Standard Generalized Markup Language (SGML) [63], an international standard. SGML is a language based on the generic markup of the structural elements of a document without regard to their presentation, which is regarded as a separate issue [35, 63]. One system, Chameleon [85], uses SGML as an intermediate language for translating between different markup languages. Another advantage of using a language such as SGML is that it facilitates the division of documents into self-contained fragments, for example, paragraphs. A further advantage of such markup languages is that they permit the specification of links and references within the text; for example, specification of hypertext links in SGML has been investigated by Bornstein and Riley [21].

The design of a document database system should consider how text is presented to the user. This presentation depends in part on the format in which the text is stored. The user interface must be able to translate text with descriptive markup such as SGML into a format suitable for display to a user.

A drawback of using SGML or any other markup language is that markup is typically not used for indexing. If this is the case, the database system must strip the markup before constructing its indexes.

### 1.2.2 Document Structure

There are two aspects to a document structure: *logical structure* and *layout structure*. Consider, for example, the logical and layout structures in this thesis. The logical structure is concerned with the logical relationship between elements such as chapters, sections, subsections, and paragraphs, whereas the layout structure is concerned with the positioning of elements such as pages, lines of text, and figures.

Office Document Architecture (ODA) [64] is an international standard which can describe both aspects of a document’s structure. Brown [22] describes ODA as providing a hierarchical and object-oriented document model. In ODA components of documents are called *objects* with information called *attributes* of the objects.

A document in ODA is represented by two tree structures: one representing its logical structure and the other its layout. This is useful for representing multimedia
documents. In each tree the structure of a document (the shape of its tree) is separate from the content of a document (the leaves of its tree). These two trees are illustrated in Figure 1.2, the logical structure at the top and the layout structure at the bottom. In this diagram, the content of the logical objects correspond to the content of the layout blocks; this is not always the cases, for example, when a paragraph is split across two layout pages. ODA also allows generic structures which represent document classes or “styles”.

1.2.3 Querying a document database

The kinds of queries in a document database system are more specialized than in a standard database system. Parsaye et al. [101] describe three paradigms for query formulation in information retrieval: a matching paradigm, an exploration paradigm, and a conceptual retrieval paradigm.

matching paradigm

A query in the matching paradigm involves returning a set of documents (or document parts) that in some sense “match” the query.

Queries in the matching paradigm may involve finding matching words and phrases in different portions of documents, including the following: the title of a document; the authors of a document; a set of keywords and phrases (selected manually or automatically) describing the subject of a document; an abstract (possibly machine generated) of a document; the full text of a document; and the full text of a section of a document.

When matching words in a document to words in a query we may wish to treat some non-identical words as matches, for example, the words ‘computable’ and ‘computability’. Such matches are examples of word comparisons that use transformations. Transformations on words include: stemming (the same root after removing suffixes), case mapping (do not distinguish between lower and uppercase), soundex mapping (similar sound when pronounced), and synonym mapping (the same thesaural class).
Figure 1.2: Logical and layout structures in ODA (based on a figure by Brown [22])
Within the matching paradigm Parsaye et al. [101] describe two approaches: deterministic matching via Boolean queries and similarity matching. It is possible to combine these different paradigms and approaches.

**Boolean queries**

*Boolean queries* can be on single words, phrases, and Boolean combinations of words and phrases. The result of a Boolean queries is usually an unordered set of all matching documents or fragments.

An example of a Boolean query is: find all documents by “A. Turing” with “computable” or “intelligent” in the title.

For imprecise queries neither ‘all solutions’ nor ‘first solution’ query evaluation strategies are appropriate: some documents will closely match the query whereas others will only be a poor fit. One appropriate query evaluation technique for imprecise queries is to rank retrieved items on the basis of estimated relevance to the query and return the most closely matched items in order of importance [121].

**similarity matching**

In *similarity matching* the query is a list of relevant words and phrases (possibly weighted). The result of a query using similarity matching is a ranked list of answers; the first is the answer judged the most likely to be relevant, the second the next most likely, and so on.

An example of a query using similarity matching is the list of words: “Turing”, “computable”, and “intelligent”.

*Exploration* queries are common in hypertext where a user navigates the database.

**exploration paradigm**

In the *exploration paradigm* a user directly explores the database; a query in this paradigm often involves only a single answer, that is the next screen to be displayed.

Different kinds of exploration queries include browsing within a single document and following references to other documents.
The *conceptual retrieval paradigm* is an advanced form of the matching paradigm, which is not yet fully developed. It requires using artificial intelligence techniques and is beyond the scope of this thesis.

**conceptual retrieval paradigm**

The *conceptual retrieval paradigm* involves making inferences about the concepts in the query and in matching documents. A document matches a conceptual retrieval query if the answer to the query can be inferred using the document.

### 1.3 Hyperbase systems

We introduce, in this section, an example of an application of a document database system that we shall be using throughout the thesis.

Ordinary written documents present text in a linear fashion which is most conveniently read from beginning to end. Even where adjacent parts, such as articles in a journal, are not logically ordered, an order is imposed by relative positions in the journal. An alternative approach is that of *hypertext* [34]. In hypertext text is presented in a non-linear fashion. Documents are divided into self-contained fragments which are stored in *nodes*.

**node**

A *node* is a logically self-contained text fragment, such as a paragraph or table, that is small enough for display to a user.

Text within each node is ordered to form the English sentences of the original document. A logical ordering of nodes within documents may exist, but is less significant.

Relationships between nodes are expressed by *links*.

**link**

A *link* is a connection from a *source node* to a *destination node*.

For example, a node that is from an article in an academic journal may have links to other parts of the article, links to the other articles cited in that node, and links to
nodes containing text on the same topic. In some applications it may be desirable to require that links are bidirectional; that is, if there exists a link from node \( a \) to \( b \) then there also exists a link from \( b \) to \( a \).

There may be many links between documents in a document database because a document may contain references to other documents in the database. These references could be scholarly citations in a journal article or could be references to earlier memorandums in business correspondence. The type of relationship between the source and destination nodes is called the \textit{link kind} and associated with a link is \textit{display} information concerning how the link is displayed in the source node to the user.

Hypertext provides one suitable interface for users to examine collections of text. Hypertext systems, that cater for large volumes of text, should be associated with information storage and retrieval methodologies. One straightforward way to implement a hypertext system is as an application of a document database system. A document database system with support for hypertext is called a \textit{hypertext database system} or \textit{hyperbase system}.

**hypertext database system**

A \textit{hypertext database system} (or \textit{hyperbase system}) is a document database system which supports links between nodes and allows the user to browse through the resulting complex network. A document database containing such links is called a \textit{hypertext database} or \textit{hyperbase}.

In the following sections we describe a layered architecture for such an implementation and discuss the features that the underlying document database system should support.

In a hyperbase system, users are encouraged to browse text in any sequence they choose by following links rather than in a sequence imposed by an author or publisher. A user might first request information on “computable numbers”. Following a train of thought, that user might then look up related information on “effective calculability” and the “Entscheidungsproblem”. Under hypertext, following this sequence is as easy as leafing through a series of adjacent pages in a book. In order to
facilitate this non-linear inspection of data, hypertext systems provide navigational aids including graphical browsers, history lists, maps of the data being traversed, maps showing where the current node belongs, and visual markers such as trails on maps to indicate where the user has been. The hyperbase system should also support the other query paradigms discussed in Section 1.2.3.

Hyperbase systems combine the high-level abstract models of documents and relationships between documents found in hypertext systems with the methodology for information storage and retrieval of database systems. Although we use a hyperbase example throughout this thesis, there are other applications for text document database systems, for example, authoring systems, office automation, electronic conferencing, and electronic books. We have chosen to model a hyperbase system as it is a typical application of a document database system; many aspects of its design are common to these other document database applications.

In Section 1.3.1 we describe an architecture for a hyperbase system. The document database system at the kernel of this architecture does not include that part of the hypertext system relating to the (window-based) user interface. In this thesis, we focus on document database design, so we do not address issues such as link generation and user interfaces.

Some documents contain parts which use media other than written text. For example some documents may contain, in addition to text, diagrams, still images, moving pictures, and sound clips.

**multimedia document**

A document which contains more than one medium is a *multimedia* document.

**hypermedia system**

A hypertext system which contains multimedia documents is a *hypermedia* system.

Management of multimedia documents [6] and hypermedia systems [59] are beyond the scope of this thesis since we restrict our attention to text documents.
1.3.1 An architecture for hyperbase systems

Zobel et al. [158] propose an architecture for a hyperbase system organized in three layers, as illustrated in Figure 1.3. Each layer is described by a distinct logical model and requires a query language for interrogating the database that is appropriate to the layer and its representation. Description of the entities to be stored can be provided through a nodal schema that provides conceptual modelling of a hyperbase at the node layer and presentational modelling at the interface layer. The conceptual model of the hyperbase at the node layer can be mapped to a logical database schema at the database system layer.

We now discuss the properties of each layer and of the nodal schema. We do not attempt to solve all the issues of the nodal schema definition language, the interface layer, and the node layer in this thesis. Instead we use the design of the database layer of a hyperbase system as a typical application of a document database system.

1.3.2 Nodal schema

Hyperbases can be designed at the node layer, via a nodal schema which describes the kinds of nodes and links held in the system, and how they interact.

A nodal schema consists of a specification of node classes and classes of links
between nodes. The specification of a node class includes information such as data to be found in nodes of that class and the classes of links to and from nodes of that class. For example, a node containing text from an Act of Parliament could contain that text, the title of the Act, and links to other Acts cited in that text. The specification of a link class would include information such as the classes of nodes linked and the semantics of the link class; for example, a link between conceptually adjacent pieces of text, such as consecutive sections of an act, would be a next/previous link.

A conceptual representation of the stored data, as specified by the nodal schema, serves several purposes. First, user queries will be formulated with regard to this representation. Second, it will be used to determine how the data will be presented to the user, and to generate maps and other navigational aids. Third, it will be used to help determine the form in which the data will be stored at the database level; physical schemas, indexing, and data representation will be generated from nodal schema models. Fourth, it will guide transformation of data between layers, including transformation of user requests into node queries, of node queries into database queries, of stored data into nodes, and of nodes into graphical objects.

A complete definition of a model for nodal schemas is beyond the scope of this thesis. Instead we use an informal description of a nodal schema and examine choices at the database layer for implementing it.

1.3.3 Interface layer

The top layer of the architecture is the user interface. Users will interact with hyperbases via natural language and graphical requests. For example, a user might be presented with a graphical representation of a node as shown in Figure 1.4. This node is a portion of the article *On Computable Numbers, with an Application to the Entscheidungsproblem* by Turing [145]. Links to related material are printed in bold type and we refer to these links as hyper links. Links to adjacent nodes are shown as buttons at the bottom of the window. A natural language request might be to request nodes related to this node that also discuss computers. A graphical request might be to select “A. M. Turing” and follow the hyperlink to nodes describing his
The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable numbers, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

Part of the function of the interface layer is to transform user requests into node queries. Thus natural language understanding should be available at the interface layer. Such understanding would require contextual and semantic information concerning the data in question that would have to be supplied from the layers below.

Another function of the interface layer is to transform nodes into graphical objects for display to a user. The appearance of this layer need not be explicitly specified by the system administrator, but could be a function of the conceptual design of the node layer as specified in the nodal schema. Thus a node could be presented as a window as shown above. Each link from the node could be represented as a button, a section of highlighted text, or, if there were several links of the same kind, they could be made available in a menu. If the text in the nodes contained descriptive markup (as discussed in Section 1.2.1) the system would use the markup to transform the data into a form appropriate for display to a user. Exactly how a node is displayed depends on the markup embedded in the node's text and should as far as possible be independent of the user's environment. Quite different views
of the data are possible by using different applications at the interface layer.

The detail of the interface layer is beyond the scope of this thesis and to simplify our description of the lower layers we omit descriptive markup details from all further examples in this thesis.

### 1.3.4 Node layer

The node layer includes the nodal schema and a node query language for interrogating the stored data. At this layer, data is organized into nodes and links, that is, into hypertext. In contrast to the interface layer, where data exists only as graphical objects, nodes and links are structured, typed data objects. In contrast to the database layer, nodes and links are abstract data objects whose form is independent of the storage organization and physical representation.

The node query language should provide the kind of access to the node layer that a conventional query language provides to a conventional database. The language needs types that allow manipulation of nodes, links, and the text, Boolean structures and other conditions that modify the set of solution nodes. To provide contextual information the hyperbase requires objects such as the currently active node and a list of previously visited nodes. As well, it needs a set of functions of sufficient expressive power to make possible the sorts of queries a user may wish to make. These must allow the selection of nodes related to other nodes, allow following links, and allow ranking and selection of nodes on the basis of particular conditions. Fuller et al. [53] discuss such a language.

At the node layer, hyperbase systems resemble object-oriented database systems. However, nodes and links are highly specialized kinds of objects: nodes often hold textual data, the structure of which is difficult to model with existing formal modelling techniques, and access to nodes and links is provided by informal as well as formal queries.

As discussed above, the nodal schema can be used to guide transformation of node queries into database queries, whose exact form will depend on the database schema generated from the nodal schema. Similarly, the nodal schema can be used
to guide transformation of data retrieved from the database into nodes and links.

As well as the nodal schema and the node query language, the node layer also needs procedures to allow various activities to take place. The layer must be able to receive nodes and links from an input application. It must have a method for informing the database layer of the appropriate way to insert the data in the nodes in the database. It must also have a method that allows for the results of queries on the database to be coalesced for the results to be displayed.

A large part of the node layer may be implemented directly by using an appropriate database system at the database layer. In Chapter 2 we model the data objects of the node layer using a variety of data models for the database layer.

1.3.5 Database layer

The database layer is the bottom layer of the hyperbase system architecture. There are several requirements that the document database system must satisfy. First, and most importantly, the document database system must have support for retrieval of text based on its content. This involves indexing on the individual words occurring in the text; such indexes are usually implemented as inverted files [156] or as signature files [67, 121]. Second, the document database system should either embed ranking techniques in the database system or provide the information needed for ranking to the node layer. Implementing ranking algorithms at the node layer is possible, but without substantial kernel support is significantly slower, because efficient ranking algorithms use statistical information stored within indexes in the kernel. Third, because of the nature of text, the document database system should be able to store variable-sized entities. Fourth, it would be useful to be able to store nodal schema in the document database system, either as data entities or in the data dictionary. Fifth, the document database system must be directly accessible from a programming language, preferably the programming language in which the rest of the hyperbase system is implemented.

The primary advantage in separating the node layer from the database layer is that an existing database management system can be used; this significantly reduces
the implementation effort, makes the system more robust, and allows traditional
database applications and hyperbase applications to share data more easily. A
disadvantage in having separate layers is probably less efficient performance; building
a hypertext system on top of an existing database management system may result
in overheads due to the interface between the systems. Nevertheless, the reduced
implementation and maintenance effort as well as the separation of functionality are
convincing arguments for having separate layers.

1.4 Overview of thesis

In the following chapters of this thesis we address issues of physical and logical design
of document databases and document database systems. These issues include the
choice of data model, the query language requirements, physical schema design,
logical schema design, and an underlying model of text. In examining these issues
we use the hyperbase example introduced in Section 1.3.

Features and inadequacies of some existing data models for representing docu-
ment databases are identified in Chapter 2. We describe two data models based on
complex objects. One model is a record-based model, used by the Atlas database
system developed at RMIT and The University of Melbourne. We use this model to
describe physical storage structures in later chapters. The other model is a higher-
level object-based model which can be used to describe the logical structure of the
database as seen by users, application developers, or database administrators.

Chapter 3 reviews the development of database query languages, especially those
for nested relational and complex object database systems. TQL has some advan-
tages as a low-level language for retrieval from document databases. TQL queries
can be directly made on both data models for complex objects described in Chap-
ter 2. The mapping of queries from the logical to the physical level is also addressed.
Possible higher-level query languages designed specifically for querying document
databases are considered.

In Chapter 4 we are concerned with schema design, both logical and physical,
using complex object models. We describe a mapping into our logical model from
entity-relationship diagrams, and a mapping into our physical data model.

In order to compare the efficiency of different structures for document databases it is desirable to have a reasonably accurate model of document text. A common model of the distribution of words in text is the Poisson approximation to the binomial distribution. A shortcoming of the Poisson model is that it ignores the effect of clustering — the property that if a word occurs at least once in a document then it is likely to occur again. Our analysis of several document collections shows that the Poisson approximation can significantly overestimate the probability that a document contains a word. In Chapter 5 we propose a new model for distribution of words in text, and show how this model can be used to estimate the probability that a document contains a word and the number of distinct words in a document.

Chapter 6 examines the design of efficient storage structures for document databases; we are particularly interested in the effect of partitioning large documents into fragments. The formulas describing the efficiency of the different physical structures use the model analysed in Chapter 5.

The final chapter, Chapter 7, summarizes the problems addressed in the thesis, some solutions to these problems, some limitations of our approach and areas for further research.
Chapter 2

Data Models

The data model or models used during the design process, and in the database implementation, are fundamental to any methodology for the design of document databases. In this chapter we survey the development of data models from the 1960s to the 1990s, that is from the classical data models, (hierarchical, network, and relational) through semantic data models to current developments in object-oriented and deductive data models. The focus of our survey is the applicability of each data model to the design of document database systems; our goal is to determine which modelling features are needed in document database design.

We describe the main aspects of data modelling in Section 2.1. In data modelling, a data model is used to describe a database schema for a particular application. In Section 2.2, we describe a database schema for our hyperbase example using the entity-relationship model, a widely used semantic data model. In subsequent sections, we present similar schemas for our hyperbase example using different data models. In Section 2.3, we use the classical data models: the hierarchical, the network, and the relational data models. In Section 2.4, we consider how extensions to the relational data model could be applied to our example; these extensions include a nested relational data model, a nested sequences of tuples data model, a deductive data model, and an object-oriented data model.

We describe, in Section 2.5, two other data models suitable for modelling document databases, one at the physical level and the other at the logical level. Both
these data models are based on complex objects; the physical data model is used by
the Atlas system, and we propose a corresponding logical data model. These two
data models are used throughout the rest of this thesis.

In the final section, we summarize the various approaches to modelling a docu-
ment database system.

2.1 Data models and database schemas

The purpose of data modelling, as described by Tsichritzis and Lochovsky [143], is to
develop models of real-world situations that are accurate and amenable to computer
representation. In database applications, real-world situations are modelled using
a consistent formal set of “general statements about how data are organized and
processed” [143] called a data model. Korth and Silberschatz [75] identify four
elements of a data model, which we use as the basis for our evaluation of different
data models.

data model

“A data model is a collection of conceptual tools for describing data, data
relationships, data semantics, and data constraints.” [75].

Abiteboul and Hull [4] also include a data-manipulation component as part of the
data model. We consider query languages, that is the data-manipulation component,
in Chapter 3.

Different data models provide different levels of abstraction. Some are close to
the actual physical representation; others provide higher levels of abstraction. Thus
we distinguish between physical and logical data models.

physical data model

A physical data model is a low-level data model which provides “concepts that
describe details of how data is stored in the computer” [46].
A logical data model is a high-level data model which provides “concepts that are close to the way users perceive the data” [46].

Elmasri and Navathe observe that high-level data models “are sometimes called object-based because they mainly describe objects and their interrelationships” [46], whereas lower-level data models “are sometimes called record-based” because “they represent data using record structures” [46]. Kent [71] discusses the limitations of record-based models for representing information, and argues the need for higher-level semantic models.

Some data models, such as the classical data models, are sometimes used to describe aspects of both the logical and the physical design of a database.\footnote{Elmasri and Navathe [46] suggest a third intermediate form of data model which they describe as an implementation data model. It provides “concepts that may be understood by end-users but that are not too far removed from the way data is organized within the computer” [46].}

Using the concepts in a data model, we can describe the structure and constraints of a database for a particular real-world situation. This description is a database schema; it is specified when a database is designed and it only changes infrequently [46].

A database schema is a description of the structure of a database together with the constraints on the data that can populate that structure. A database schema is also an intension of the database.

The part of a database that we do expect to change frequently is the stored data. In a hyperbase, for example, we may add documents or links between documents, we may delete existing documents or links, or we may edit an existing document.
database state

“The data in the database at a particular moment in time” [46] is a database state. A database state is also a database instance or an occurrence, or an extension of the schema.

A database schema defines a set of legal database states.

integrity constraints

The static integrity constraints define which states are legal database states, and the dynamic integrity constraints define which transitions between states are legal.

Elmasri and Navathe [46] distinguish between three types of static or dynamic integrity constraints: constraints that are inherent in a particular data model, implicit constraints that are supported by the data definition language of the data model, and explicit constraints that are not supported by the model and must be checked in applications. Most database management systems provide a capability to represent a range of static constraints, but allow few, if any, dynamic constraints.

Proposing new data models has been a popular research field for many years. Beeri [13] discusses the evolution from relational to object-oriented models. Schek and Scholl [126] discuss the evolution of the more recent object-based data models from several earlier data models. Expanding on Schek and Scholl’s classification, we show in Figure 2.1 the evolution of the main families of data models. We show particular data models at the approximate year in which they first appeared, that is either when the data model was proposed or when a system embodying an implementation of the data model was released. In the case of semantic data models, we show them first appearing in 1974 (with the binary-relationship model of Abrial [5]), but do not show all the subsequent versions (such as the entity-relationship model of Chen [27], the NIAM model of Falkenberg [48] and Nijssen [95, 96]). The arrows in the figure denote the development of concepts from earlier data models to later data models; for example, the nested relational model combines the efficient tree structures of the hierarchical model with the elegant mathematical basis of the relational model.
2.2 Semantic data models

We begin our review of data models with a group of high level logical models called semantic data models. Borkin [19, 20] distinguishes semantic data models from other data models, which he calls syntactic. Syntactic data models, such as the classical data models, allow “arbitrary syntactic structures such as trees, networks, or relations” whereas semantic data models attempt “to provide users of the data model with a clear interpretation of the database in terms of the relevant application” [19]. Hull and King [62] argue that semantic data models have three main advantages over classical record based models: a greater separation of logical and physical schemas, less semantic overloading of relationships, and more convenient abstraction mechanisms [62].

2.2.1 Entity-relationship model

A widely used semantic data model is the entity-relationship model (or E-R model) described by Chen in 1976 [27]. This model and variants of it have been described in many database texts [42, 46, 75, 143, 146]. There have been many proposed extensions to the E-R model and many descriptions of the model include some of
these extensions. For example, Parent and Spaccapietra [100] enhance the E-R model to allow attributes to be complex objects. Another extension is the extended higher-order E-R model described by Thalheim [135]. We use the enhanced E-R model, described by Elmasri and Navathe [46]. This model incorporates concepts from some of the more recent semantic data models.

The basic concepts used in the E-R model are attributes, entities, and relationships. Corresponding to attributes, sets of entities, and sets of relationships are graphical symbols which can be combined to form an entity-relationship diagram (or E-R diagram) that is a pictorial representation of an E-R schema. A possible E-R schema for a hyperbase is shown in the E-R diagram in Figure 2.2.

entity

An entity is an object that exists in the world that we wish to model and is distinguishable from other objects. Entities can be either concrete or abstract. Entities which are similar are grouped together; an entity set is a set of entities of the same kind. In E-R diagrams entity sets are shown using rectangles that are labelled with the name of the entity set.

An example of an entity is the hypertext node displayed in Figure 1.4 of the previous chapter. We choose to represent a set of similar hypertext nodes by the entity set node, and we represent a set of document entities by the entity set document. This choice of entity sets for representing a hyperbase is shown in the E-R schema in Figure 2.2. An alternative representation is that links, as well as nodes and documents, could be represented by an entity set, Kent [70, 71, 72] provides an in-depth discussion of the problem of deciding which things are entities and identifies the limitations of record based data models.

Each entity in the world being modelled is described by a number of attributes.
Figure 2.2: An E-R diagram for a hyperbase
attribute

An attribute is a property of an entity or a relationship. Associated with each attribute is a domain (or data type) which is the set of permitted values. “Formally, an attribute is a function which maps an entity set into a domain” [75]. In E-R diagrams attributes are shown using ellipses; each ellipse is labelled with the attribute’s name.

Entities in the entity set document have the attributes docid and title. Some attributes, such as a person’s name, have several parts.

composite attribute

A composite attribute is an attribute that itself has attributes.

For example, a composite attribute representing a person’s name may have attributes firstname and surname.

Most attributes are have simple domains; however, some attributes are set valued, that is such attributes have a domain that is a powerset of some simpler domain.

multivalued attribute

A multivalued attribute is an attribute that has a set of values. These are shown in E-R diagrams using concentric ellipses.

For example, author is represented as a multivalued composite attribute comprising a set of firstname, lastname pairs. An alternative representation would have author as a separate entity.

Entities are identified by a unique entity key.

document

A set of attributes which uniquely identify an entity in an entity set is an entity key. These are shown in an E-R diagram by underlining the attributes forming the key.

For example the attribute docid is the key of the entity set document. Some entity sets do not have a key which is any subset of their attributes; their identity depends
upon another entity set to which they are related. Such entity sets are weak entity sets and are shown in E-R diagrams using double boxes. There are no examples of weak entities in our example hyperbase schema.

Entities can be related to one another via relationships.

**relationship**

Two or more entities, \(e_1, e_2, \ldots, e_n\), can be associated to form a relationship. Just as similar entities are grouped together into entity sets, similar relationships are grouped into relationship sets. “A relationship set is a set of relationships of the same type. Formally, it is a mathematical relation on \(n \geq 2\) (possibly non-distinct) entity sets. If \(E_1, E_2, \ldots, E_n\) are entity sets, then a relationship set \(R\) is a subset of

\[
\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}
\]

where \((e_1, e_2, \ldots, e_n)\) is a relationship” [75]. In an E-R diagram a relationship set is represented by a labelled diamond that is connected by line segments to the entity sets participating in the relationship set.

There are relationships between the node in Figure 1.4 and each reference to another node, such as the buttons **next**, **parent**, and **previous** displayed at the bottom of the node. A relationship set between two sets of entities must be one of the following: one-to-one (shown by 1:1 on edges), one-to-many (1:N), many-to-one (N:1) or many-to-many (N:M). An example of a one-to-many relationship set is contents between document and node. Each entity set may play one or more roles in a relationship set. In the example the entity set node plays two roles, from and to, in the many-to-many relationship set link.

Entity sets are not always distinct. In a hypermedia system containing both text and pictures two special types of nodes are required: **textnode** (nodes containing text) and **picnode** (nodes containing pictures). Both the subtypes **textnode** and **picnode** are subsets of the more general supertype **node**. These relationships are ISA relationships. For example, a **textnode** ISA node.
An example of an ISA relationship is shown in Figure 2.3 where the supertype node is specialized into the subtypes textnode and picnode. The circle containing D indicates that these are disjoint subtypes. Some authors distinguish between two different types of ISA relationships: generalization and specialization. A (free) subtype can be defined as a specialization of an existing supertype, a (free) supertype can be defined as a generalization of an existing subtype. The distinction between specialization and generalization can not be shown in the enhanced E-R diagrams which we are using.

E-R diagrams are not unique representations of a world situation they are modelling; for example, we can represent an author either as a composite multivalued attribute or as a separate entity set. The Fact model [5] mentioned in Section 2.2.2 overcomes some of these problems. Kent [72] also favours a fact-based approach.

Jajodia et al. [66] investigates the issue of equivalence of E-R diagrams. They
develop three definitions of E-R diagram equivalence. Two E-R diagrams are *domain data compatible* if the universal relation schemas constructed from the E-R diagrams can be equated. Two E-R diagrams are *data dependency equivalent* if they both generate the same set of functional dependencies over the same universal relation schema. Two E-R diagrams are *instance data equivalent* if they have the same set of fixed points of a project-join mapping of their instances. Chen [27] discusses mappings from the E-R model to the relational and network data models. Elmasri and Navathe [46] list other work done on mappings to and from the E-R model.

Although E-R diagrams are able to model a wide variety of real world situations, there are some situations which E-R diagrams can not model easily. One noticeable limitation for representing document databases is the lack of a facility for specifying that some relationship sets are ordered. For example, to represent that nodes in a document are ordered requires the addition of an attribute indicating the position number of a node within a document.

### 2.2.2 Other semantic data models

Extensive surveys of semantic data models have been compiled by Hull and King [62] and Peckham and Maryanski [102]. Other semantic data models mentioned in these surveys include IFO [3], the binary-relationship model [5], RM/T [33], SDM [61], the functional data model and data definition language DAPLEX [132], and GEM [144]. Hull and King identify a trend that “although the semantic data models were first introduced as design tools, there is increasing interest and research directed toward developing them into full-fledged database management systems” [62]. Elmasri and Navathe [46] give the following examples of the direct implementation of a semantic data model: the ZIM system by Xanthe Corporation which is based on the entity-relationship model, the SIM product of Unisys which was the first such commercial implementation of a semantic data model, and possibly the entire range of object-oriented database systems.

The simplest fact-based model is the binary-relationship model, which was originally proposed by Abrial [5] in 1974. It was the first data model that could be
described as semantic. In the binary-relationship model the basic concepts are categories and binary relations. The binary-relationship model is similar to the E-R model except that no distinction is made between attributes and entities (both are categories), and only binary-relationships are permitted. A variation on the binary-relationship model is to allow $n$-ary relationships as well as binary-relationships. Fact-based models which allow $n$-ary relationships include the NIAM Conceptual Model [77, 78, 96, 97] and the Fact model [115]. NIAM diagrams [77, 96, 97] can be used to represent NIAM conceptual schemas.

Leung and Nijssen [77] describe a procedure for transforming a NIAM conceptual schema into a relational schema in Optimal Normal Form. Optimal Normal Form is a normal form between project-join normal form and domain-key normal form (see Section 2.3.3). These transformations have been incorporated in the software product RIDL developed by Meersman et al. at Control Data.

Object-oriented extensions to the binary-relationship model — incorporating encapsulation, specialization, type constructors and behaviour — are proposed by De Troyer [44].

A deductive database extension to NIAM, based directly on logic, called Logic Oriented Conceptual Schema (LOCS) is described by Dart and Zobel [39]. Associated with LOCS is graphical query language called Piq. One feature of LOCS is that it is possible to express arbitrary constraints in first order logic.

Although the binary-relationship model and its variations avoid the problem of choosing which things are entities and which are attributes, the binary-relationship model does this at the cost of the complexity (that is, number of arcs) in the resulting diagrams.

### 2.3 Classical data models

The three classical data models (hierarchical, network, and relational) were conceived in the 1960s. Each of these data models are record oriented; so they are still close to the actual computer implementation but provide some abstraction of the data.
2.3.1 Hierarchical data model

The hierarchical model, as the name suggests, is based on hierarchical (tree) structures of records. For many applications this is a particularly efficient storage structure. IMS (Information Management System) is the most notable example of a database management system based on the hierarchical model. The first version of IMS, one of the first commercial database management systems, was released by IBM in 1968. The importance of the hierarchical model is reflected in the fact that IMS still ranks in the commercial market as the dominant system for supporting large-scale accounting and inventory systems [46].

In the hierarchical data model a record is “a collection of field values” [46]. A collection of records with the same structure and purpose are grouped into a record type.

record type

A record type has a name and a structure. The structure is a collection of named fields called data items.

The hyperbase example, shown in the E-R schema of Figure 2.2, could be implemented using a hierarchical data model as shown in Figure 2.4. In this example, the record type named document has the fields named docid and title as data items.

Relationships between record types in hierarchical data models are represented using the parent-child relationship type.

parent-child relationship type

A parent-child relationship type (PCR type) is a one-to-many relationship between a parent record type and a child record type. “An occurrence (or instance) of the PCR type consists of one record of the parent record type and a number of records (zero or more) of the child record type” [46]. The set of child records in an occurrence is logically ordered.

Between the parent record type document and the child record type node in Figure 2.4 is a PCR type indicated by the solid line on the diagram.
One or more PCR types are arranged in a tree structure called a *hierarchical schema*. A *hierarchical database schema* consists of one or more hierarchical schemas.

With a purely hierarchical structure there are some problems [46]. There is the problem of how to represent many-to-many relationships and N-ary relationships. Furthermore, child record types that appear in more than one PCR pose a problem. The problems of many-to-many relationships and multiple child record types can be avoided with duplication of data. However, a more desirable solution is virtual (or pointer) record types [46].

**virtual record type**

A *virtual (or pointer) record type* “is a record type with the property that each of its records contains a pointer to a record of another record type” [46].

Pointers to virtual records are shown in dotted boxes with arrows to the record being pointed to; for example, in Figure 2.4, the virtual parent type `link` points to...
the virtual child type node.

There are a number of inherent constraints in the hierarchical data model between child and parent records [46]. Only root records can exist without being related to a parent record occurrence. If a child record has more than one parent record from the same record type, the child record must be duplicated. A child record can have multiple parent records of different record types but only one real parent but multiple virtual parents.

Where the data being modelled has mostly hierarchical relationships the hierarchical model is good but where there are many non-hierarchical relationships the hierarchical model is unsuitable. There is not a unique mapping from the E-R model to the hierarchical model, as there are a number choices that can be made. Elmasri and Navathe [46] show that M:N relationship types can be represented in more than one way. An M:N relationship type can be represented as a 1:N relationship type by duplicating records at the N-side when a record is related to more than one parent. Alternatively, it can be represented by more than one hierarchy and using virtual records.

An advantage of using the hierarchical model for representing document databases is that documents usually have a hierarchical structure; for example, both the logical and layout structures defined in ODA are hierarchies. Furthermore, occurrences of parent-child relationships are ordered, which reflects the logical sequence of the contents in most documents. A disadvantage of the hierarchical model is the lack of an ad hoc query language.

### 2.3.2 Network data model

According to Elmasri and Navathe [46] the origin of the network model was GE’s Integrated Data Store designed by Bachman in 1961. The model was developed and formalized by the CODASYL Database Task Group in 1971 with revisions in 1978 and 1981.

The example schema from Figure 2.2 could be implemented using a network data
Figure 2.5: A Bachman diagram for the network representation of the hyperbase model as shown by the *Bachman diagram* in Figure 2.5. Elmasri and Navathe give a mapping from the E-R model to the network model.

In the network model a record stores a group of related data values.

**record type**

A *record type* describes the structure of a group of records that store the same type of information. Each record type is given a *name* and each *data item* or attribute in the record has a *name* and a *format* (or data type).

Examples of record types in the Bachman diagram in Figure 2.5 are *document, node, link, author*, and *authors*.

A *vector* (or single multivalued attribute) is a data item that can have multiple values in a single record. A *repeating group* (or composite multivalued attribute) is a data item that can have a set of composite values in a single record.
An actual data item is the value actually stored in a record as distinct from a virtual (or derived) data item which is not stored but defined in terms of actual data item/s.

**set type**

A set type is a description of a 1:N relationship between two record types an owner record type and a member record type. A set occurrence (or set instance) comprises one owner record from the owner record type together with zero or more of related member records from the member record type.

These are not sets in the mathematical sense because there is one distinguished element (the owner record) and the member records are ordered, thus these are sometimes called ownership coupled sets or co-sets. Examples of set types in the Bachman diagram in Figure 2.5 are contents, from, to, written by, and wrote with.

A number of constraints are inherently enforceable by the network model. For example, an inherent constraint that every network schema must obey is that “a record of the member record type cannot exist in more than one set occurrence of a particular set type” [46]. Other constraints can be expressed in the network data model, for example whether member records can exist on their own or must be part of a set occurrence.

The network model retains the advantages of a hierarchical model, and does not have the restriction that a child record can only have one real parent. This makes N:M relationships easier to model, which is useful for representing links between nodes. The network model shares the limitations of the hierarchical model in that ad hoc queries against a document database are difficult for users to express because the network model requires a “record at a time” data manipulation language.

### 2.3.3 Relational data model

The relational model developed by Codd [32] in 1970 has since been the basis of much database research and development. Its wide acceptance was due to the simple
but formal nature of the mathematical relations and relational query languages. The relational model is based on the formal mathematical concept of a relation.

relation

A relation is a subset of the Cartesian product of the domains of its attributes. If a relational schema has \( n \) attributes \( A_1, \ldots, A_n \) then a relation (or instance of the schema) is a set of \( n \)-tuples.

Elmasri and Navathe [46] give a mapping from the E-R model to the relational model. The example schema from Figure 2.2 could be implemented using a relational data model as shown in Figure 2.6. In this figure we enclose the attributes of relational schemas document, author, node, and link with ‘[’ and ‘]’. Thus docid, title are attributes of the relation document. In this thesis we use the following conventions for writing schemas and queries: keywords (such as the domain names INTEGER and TEXT) are shown in uppercase; table and attribute names are shown in lowercase.

key

A key is a minimal set of attributes from a relational schema that uniquely identify an \( n \)-tuple in the relation.

In this schema we only show key and domain constraints. A key is shown for each of the relations in Figure 2.6. A foreign key is an attribute that is the key to another relation. These are not shown in the schema in Figure 2.6; however, the attribute ofdoc in the node relation and the attribute docid in the author relation are foreign keys as they are both keys to the relation document. Similarly, the attributes nodefrom and nodeto are foreign keys as both attributes must have values drawn from the key of the relation node.

Many other types of constraints are possible: for example, functional dependencies, and join dependencies.
<table>
<thead>
<tr>
<th>Table</th>
<th>Field(s)</th>
<th>Data Type(s)</th>
<th>KEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>document</td>
<td>docid, title</td>
<td>INTEGER, STRING</td>
<td>(docid)</td>
</tr>
<tr>
<td>author</td>
<td>firstname, surname, docid</td>
<td>STRING, STRING, INTEGER</td>
<td>(firstname, surname, docid)</td>
</tr>
<tr>
<td>node</td>
<td>nodeid, ofdoc, data</td>
<td>INTEGER, INTEGER, STRING</td>
<td>(nodeid)</td>
</tr>
<tr>
<td>link</td>
<td>nodefrom, nodeto, lkind, display</td>
<td>INTEGER, INTEGER, STRING, STRING</td>
<td>(nodefrom, display)</td>
</tr>
</tbody>
</table>

Figure 2.6: A relational schema for a hyperbase
functional dependency

Let $U$ and $V$ be sets of attributes on a relational schema. The functional dependency $U \rightarrow V$ holds on the schema if every pair of tuples that have the same values on the $U$ attributes also have the same values on the $V$ attributes [47].

To avoid storing redundant information, and hence possibly having update anomalies, it is desirable to have a relational database normalized.

Boyce-Codd normal form

If every functional dependency in the database is a consequence of just the key constraints then the database is in Boyce-Codd normal form [47].

join dependency

A schema $R$ obeys a join dependency $\bowtie (R_1, \ldots, R_n)$ if every instance $r$ of $R$ is the natural join of its projections $\pi_{<R_1>}(r) \bowtie \ldots \bowtie \pi_{<R_n>} (r)$ [46].

project-join normal form

A schema is in project-join normal form if there are no join dependencies that are not a logical consequence of the keys [47].

A relational schema in project-join normal form, and has less anomalies than a schema in Boyce-Codd normal form.

There exist more general dependencies that are not a consequence of the keys, for example a template dependency [120] are not in all cases a consequence of the keys.

template dependency

A template dependency is defined by one or more hypothesis tuples that are examples of tuples that may appear in one or more relations, together with one or more conclusion tuples that must exist in the relations.

The following example of a template dependency specifies that if there exists a link from node $X_1$ to node $Y_1$ then there must be a link from node $Y_1$ to node $X_1$. 
domain-key normal form

A schema is in *domain-key normal form* [47] if there are no dependencies that
are not a logical consequence of the domain and key dependencies.

A relational schema in domain-key normal form has less anomalies than a schema
in project-join normal form.

The relational model overcomes the problems of the hierarchical and network
models with respect to ad hoc queries but has some other disadvantages. One
disadvantage of the relational model is that the information about a single entity
such as a document can be distributed over several tables in the relational design.
If the physical design reflects the logical schema definition (as is almost always the
case in relational databases) then the data will be stored in four physical tables; at
query time these separate physical tables will frequently be required to be joined
together to form the original entity. Another disadvantage is that because relations
are mathematical sets it is not possible to represent the logical order of the contents
of a document without using additional attributes.

2.4 Extensions to classical data models

2.4.1 Nested relational data model

Nested relational databases are part of a technology to support advanced applica-
tions; they combine the efficiency of pre-relational systems with many of the ad-
vantages of the relational approach. 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 valued 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.

The original not-necessarily-normalized relations as proposed by Makinouchi [84] simply allowed relation-valued attributes as well as atomic-valued attributes. Databases allowing relation-valued attributes and other complex objects are called nested relational databases or non-first normal form databases.

Later schemas allowed other kinds of complex objects as well, such as multisets and lists; for example, as supported in the NF² database system described by Pistor and Traunmueller [105]. A number of database research projects have been developed based on nested relations and other similar complex objects [37, 104, 110, 111, 124, 127, 128, 136]. These research projects include the AIM project at IBM Heidelberg Scientific Centre in Germany [37, 104], the VERSO project at INRIA in France [128], the DASDB project at Darmstadt in Germany [124, 125], which is the basis of the COSMOS project at ETH in Zurich, Switzerland [127], the work SQL/NF [110] and formal query languages [111] at the University of Texas at Austin, and the Atlas project in Melbourne [136]. In Chapter 3 we discuss query languages for some of these systems.

The example schema from Figure 2.2 could be implemented using a nested relational data model as shown in Figure 2.7. In this figure we enclose the attributes of tables and nested tables with ‘[’ and ‘]’. Thus firstname is an attribute of the nested table authors in the outer-level table document.

Extensions to the nested relational data model provide a variety of models for complex objects. For example, allowing nested tables which are lists as well as sets, and the addition of reference attributes. Such extensions can be the basis for an implementation of a object-oriented database system.

A problem with using nested relations at the logical level, is that we must choose which entities to nest inside other entities. We address this issue further in relation to complex object logical design in Chapter 4 and physical design in Chapter 6. One
Figure 2.7: A nested relational schema for a hyperbase
property that should be maintained in any design of nested relations is *partitioned normal form*.

**partitioned normal form**

A nested relational database schema is in *partitioned normal form* [111], if the key for each relation and nested relation schema is dependent only on simple attributes that are not nested relations.

Keys which are on simple attributes are much easier to maintain than keys involving attributes which are themselves nested tables.

The nested relational model retains the mathematical basis of relational model and the ability to support ad hoc queries while allowing a degree of data structuring. This model does not, however, support ordering of document contents as is possible using the hierarchical and network models. This problem can be overcome by extending the model to support sequences rather than sets.

### 2.4.2 Nested sequences of tuples data model

The nested sequences of tuples (NST) data model was developed by Güting and Zicari as a data model appropriate for office documents [57, 58]. The basic constructs in the data model are the *tuple* construct and the *sequence* construct.

**tuple**

A *tuple* $t$ is represented as $t = (t_1, \ldots, t_r)$ where $t_1, \ldots, t_r$ are the components of the tuple.

**sequence**

A *sequence* $s$ is represented as $s = \langle s_1, \ldots, s_n \rangle$; the empty sequence is represented by $\emptyset$.

In the NST model a schema describes the structure and possible values of a group of objects called a class. Schemas are either *atomic* or *composite*. 
CHAPTER 2 — DATA MODELS

atomic schema

The set of possible values of objects of an atomic schema is:

\[ \text{instances}(s) = \text{dom}(s) \]

where \( \text{dom}(s) \) is the set of possible values in the one of the elementary data types \( s: \) INTEGER, REAL, BOOLEAN, or STRING.

composite schema

Given the schemas \( s_1, \ldots, s_r \) the tuple \((s_1, \ldots, s_r)\) is a composite schema. The set of possible values of objects of a composite schema \( s = (s_1, \ldots, s_r) \) is:

\[
\text{instances}(s) = \{ <a_1, \ldots a_n> | n \geq 0 \\
\text{and for } 1 \leq i \leq n: a_i = (a_{i,1}, \ldots, a_{i,r}) \\
\text{and for } 1 \leq j \leq r: a_{i,j} \in \text{instances}(s_j) \}
\]

“Any value associated with a composite schema is a sequence of tuples” [57].

data object

A data object is given by a pair \((s, o)\) where \( s \) is a schema and \( o \in \text{instances}(s) \)

The formal NST model does not contain labels for fields, although for readability they are sometimes added as shown in Figure 2.8. In this figure there is a composite schema document.

The ability to represent the order of tuples in nested tables is an important advantage of the NST model over the nested relational model described in the previous section. Since nested sequences of tuples are ordered, we are able to represent a document’s contents as a sequence of node identifiers which reflects the linear order of nodes in the printed document. Similarly we are able to represent the authors of a document as a sequence of names which reflects the ordering of authors on the original document.

The NST model has all the advantages of the nested relational model and the ability to support a logical ordering for document contents. It is likely that the next generation of nested database systems will be based on the NST model rather than the nested relational model. We discuss an algebra for nested sequences of tuples in Chapter 3.
Figure 2.8: An NST schema for a hyperbase
2.4.3 Deductive data model

Deductive databases are based on first order predicate logic and extend relational databases to include rules as well as facts [54, 55, 80]. Deductive databases use these rules to deduce new facts from the facts stored in the database.

Mathematical logic, as used in logic programming [81], provides a uniform framework for the expression and manipulation of facts and rules. The basic building blocks of facts and rules are terms, atoms, and literals.

term

A term is either a variable, a constant, or a function symbol applied to other terms.

atom

An atom is of the form \( p(T_1, \ldots, T_r) \) where \( p \) is a predicate symbol and \( T_1, \ldots, T_r \) are terms.

For example,

\[
document('Intelligent machinery', \text{author('Alan', 'Turing')})
\]

is an atom, where \text{document} is a predicate symbol and \text{author('Alan', 'Turing')} is a term.

literal

A literal is either an atom (a positive literal), or an atom preceded by \( \neg \) (a negative literal).

Rules are represented using Horn clauses.

Horn clause

A Horn clause is of the form \( P : - Q_1, \ldots, Q_n \). In this rule, the atom \( P \) is the head and the conjunction of literals \( Q_1, \ldots, Q_n \) is collectively the body.

Facts are a special case of a Horn clause without a body and no variables; the tuples in a relational database are equivalent to the facts in a deductive database.
A rule without a head is a query (clause) and written $?-Q_1,\ldots,Q_n$. For example, the following query finds documents by Alan Turing.

$$?- \text{document}(X, \text{author('Alan','Turing'))}$$

The deductive database model has been used as a data model for a number of research prototypes including the MU-Prolog [90, 92] and NU-Prolog [91, 93, 106, 138] deductive databases.

A limitation of existing deductive databases for representing document databases is that they have not been developed specifically for text nor for directly supporting hierarchical structures. In the future deductive databases may play an important role in document database systems in the area of conceptual retrieval which involves natural language processing. Chat-80 [148, 149] is a natural language question-answering system which uses a small deductive database, and provides one step towards conceptual retrieval from document databases.

### 2.4.4 Object-oriented data models

During the 1980s there was a move to directly implement logical data models by way of complex object data models. Object-oriented data models are seen by Schek and Scholl [126] to have evolved from the classical data models by incorporating many of the key concepts from semantic data modelling and nested relational data models.

In the same way that deductive databases evolved out of the marriage of databases and logic programming, so object-oriented databases evolved out of the marriage of databases and object-oriented programming. As yet there is no universally accepted object-oriented data model although there is a clear need for standardization in this area [52]. Nevertheless, in some sense all logical models are object oriented rather than record (or value) oriented. Hull and King make the following distinction between semantic data models and object-oriented languages: “essentially, semantic models encapsulate structural aspects of objects, whereas object-oriented languages encapsulate behavioral aspects of objects” [62].
According to Atkinson et al. [7] there are thirteen mandatory features that a system must have to be a truly object-oriented database system. Five of these features are database requirements: persistence, secondary storage management, concurrency, recovery, sharing data between users, and ad hoc querying. Eight are object-oriented requirements: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility, and computational completeness.

Many of these object-oriented features are supported in the draft international standard SQL3 [65]. Some of these are illustrated using our hyperbase example as shown in Figure 2.9.

Systems with support for complex objects provide a suitable data model for document databases. Such systems allow the structure of the documents to be modelled and allow linkages to made between documents. Some systems have been described as structurally object-oriented in that they provide support for complex objects as distinct from behaviourally object-oriented methods which they do not include. The Atlas physical model described in Section 2.5.1 is in the structurally object-oriented category although it does not support a class hierarchy. SQL3 aims to provide both structural and behavioural object-oriented features.

\section{2.5 Complex object data models}

In preceding sections we have studied the applicability of several existing data models to problems of modelling document databases. A number of desirable features have emerged; these include complex objects with provision for nested sequences, and support for text. In Sections 2.5.1 and 2.5.2 we describe separate physical and logical data models suitable for describing document databases.

\subsection*{2.5.1 Physical object data model of Atlas}

Atlas [117] is specifically designed for applications supporting free text such as document database systems [118]. The data model supported by Atlas is a physical
CREATE TYPE author WITHOUT OID (
    firstname STRING,
    surname STRING
);

CREATE TABLE document (
    docid INTEGER,
    title CHAR(80),
    authors LIST(author),
    contents LIST(REF(node))
);

CREATE DOMAIN lkdom AS (next, previous, parent, hyperlink);

CREATE TYPE link WITHOUT OID (
    nodeto REF(node),
    lkind lkdom,
    display CHAR(20)
);

CREATE TABLE node (
    nodeid INTEGER,
    ofdoc REF(document),
    data CHAR VARYING(1760),
    links SET(link)
)

Figure 2.9: An SQL3 schema for a hyperbase
data model which is also used directly by users. The schema on which queries are expressed is used to represent physical aspects of the storage such as indexes on particular attributes. TQL is the data definition, data manipulation, and query language used by the Atlas database management system. TQL is described in Section 3.3 of the next chapter. Other features include support of text attributes, tuples, nested tables, and implicit joins using references. One way in which the hyperbase example, introduced for the entity-relationship model in Figure 2.2, could be written in Atlas is shown Figure 2.10.

The ability to support nested tables means that Atlas is able to provide a direct representation of hierarchically structured objects. This property is extremely useful in the context of document databases since objects such as documents are typically hierarchical (for example, a document consists of sections and each section consists of paragraphs). However, in a hyperbase this structure may be made flatter by using nodes and links.

Since Atlas is specifically designed to support text in document databases, text is treated as a built-in type and a number of text operators are supported. These operators support the searching of free text on a word by word basis, and the transformations of text to soundex, stemmed, and case converted forms.

Integers, floating point numbers, Boolean values and text are presently supported by Atlas as atomic attribute types. Strictly speaking, text is not an atomic type as TQL has operators which access parts of a text attribute. Values of type text are strings of free text which can be searched for words, phrases etc. Text constants are surrounded by single quote characters.

Atlas supports two structured attribute types: nested tables and nested tuples. Tables and nested tables are similar. A table is a top-level object which exists in the database and can either be global, local or temporary. The difference between global, local and temporary tables is discussed in Section 3.3.6 which describes table

---

3The Atlas database management system was formerly called Titan+. Atlas is a research prototype originally developed under GIRD funding from the Australian Government, and was developed as a successor to the commercial system Titan marketed by Knowledge Engineering Pty Ltd.
Figure 2.10: An Atlas schema for a hyperbase
creation. A nested table can only exist in the database as an attribute of a table or of a structured attribute (which could be another nested table). A nested table is an ordered list (that is, a sequence) of tuples whereas a table is an unordered collection of tuples (a multiset or bag which is distinguished from a set because it can contain duplicates). The document table contains two nested tables authors and contents. The node table contains one nested table links.

A tuple is a row (or part of a row) within a table or nested table. A composite attribute can be defined by a tuple that is a subset of the attributes in another tuple. The nested table authors contains one such composite attribute, author, which is made up of the atomic attributes firstname and surname.4

In addition to atomic attributes and structured attributes Atlas supports reference attributes. Reference attributes are described by Pistor and Traunmueller [105] and included in SQL/W [76]. A reference attribute is a tuple comprising the key of the table referenced. An alternative implementation of reference attributes is to use internal tuple identifiers. The table referenced can either be an outer level table or a nested table. Reference attributes provide support for implicit joins which are described in section 3.3.2 and support for foreign keys.

In TQL it is possible to specify keys for both outer level tables and nested tables. A key is a tuple with can contain one or more attributes which appear in the table; these attributes from the table can be either atomic or tuple attributes but not table attributes.5 Consider the table node in Figure 2.10 which specifies keys for the table node and the nested table links. The attribute (nodeid) is a key for the table node. The key (display) specified on the nested table links indicates that the attribute (display) is unique within a nested links table in one node tuple but is not necessarily globally unique across all tuples in the table node. TQL also allows composite keys and alternate keys.

4This is a slightly contrived example of a composite attribute, and could be replaced by a table with the atomic attributes firstname and surname, and no composite tuple attribute author.

5A TQL table with a key on the table and on all nested tables within it is in Partitioned Normal Form [111] since keys may not include an attribute which is a nested table.
2.5.2 Logical data model with complex objects

Although the complex objects described in the previous section provide a suitable physical model for Atlas they are not suitable for describing a logical model for document databases (or other applications for that matter). So, in this section we propose a new logical model for describing complex objects. Our proposal for logical schemas is based on the object-oriented concept of a class and is similar to the COCOON schemas described by Scholl and Schek [131] as part of the COSMOS project [127]. Our complex objects are simpler because we did not consider a type or class hierarchy and we required inverse relationships to be specified between pairs of attributes in different classes. Another difference from COCOON is that in our proposal nested tables can be lists as well as sets. Using our model for logical schemas we could represent our hyperbase as in Schema 2.11. The basic modelling concept in this data model is an object class.

**object class**

An object class is a schema describing a collection of similar objects.

In our example, document, author, node, and link are object classes. An object class is not unlike a relation schema in a relational database. Individual objects in the class are similar to the tuples in a relational database. For each class the state of the database is represented by the set of objects which belong to the class; each such set corresponds to an instance of a relation schema.

An object class can have three kinds of attributes, atomic, object-reference, multiple-object-reference attributes.

**atomic attribute**

An atomic attribute is an attribute whose type is one of the atomic types.

In the class document, docid is an atomic attribute of type INTEGER and title is an atomic attribute of type TEXT.
Figure 2.11: Database schema based on complex objects
object-reference attribute

An object-reference attribute is an attribute whose type is a single-valued instance of an object class.

For example, the attribute ofdoc in the class node is a reference to an object of class document. An object-reference attribute usually refers to an object in another class but it can refer to an object of the same class. An object-reference can be mapped into an attribute comprising a single REF at the physical level.

multiple-object-reference attribute

A multiple-object-reference attribute is an attribute whose type is a multi-valued SET OF or LIST OF references to an object class.

For example, the attribute authors in the class document is a list of references to the authors of the document, whereas the attribute wrote in the class author is a set of references to documents written by a particular author. These can be mapped into a nested table of REFS at the physical level.

The concept of one relationship being the inverse of another has been described in respect to semantic data modelling by Hammer and McLeod [61]. Although inverse relationships cannot express arbitrary constraints between sets of objects, inverse relationships occur frequently so we use the concept in our logical schemas. An object-reference or multiple-object-reference attribute A in one object class R is the INVERSE OF an object-reference or multiple-object-reference attribute B of an(other) object class S if for all pairs of objects r in R and s in S, sisamemberofπAr if and only if risamemberofπBs.

The main advantage of the logical schemas is that they have replaced the reference attributes (which can create problems with data integrity) with the object-reference and multiple-object-reference attributes. The pairs of objects containing object references are managed in a controlled way using the inverse relationship.
2.6 Summary

During the last thirty years a large number of data models have been proposed or implemented. Data models can be broadly classified as logical or physical and may be used at one or more of the three levels: external, logical, and physical.

In this chapter we have examined a diverse range of data models and illustrated these data models using a hyperbase example. Semantic data models make it easier to accurately model a hypertext style document collection. Those data models which provide complex objects allow semantic models of hypertext to be transformed in a way that is amenable to computer representation. The basis of data models supporting complex objects is the concept of nested relations.

Useful features of a data model for document database systems include complex objects with nested sequences, reference attributes, and a special type for text. We have described a physical and a logical data model which support these requirements.

In Chapter 3 we present a number of query languages which adopt the nested relational and complex object paradigms. We describe particularly the Text Query Language, TQL, a language which was designed for the Atlas physical model but can also be used with the logical model proposed for complex objects in Section 2.5.2. The Atlas schema in Figure 2.10 for a hyperbase is used to illustrate TQL. TQL can also be used for describing queries on our logical data schema. In Chapter 4 we discuss the choice of the best logical schema and describe a mapping from an entity-relationship diagram into a logical schema, and from a logical schema into a physical Atlas schema.
Chapter 3

Query Languages

Many database applications have special requirements that cannot be easily served by existing commercial relational database systems, for example, image processing, computer-aided design, office information systems, and document databases. The complex objects required for these applications can be represented by nested relations. Nested relational systems combine the efficiency of some pre-relational systems with many of the advantages of the relational approach. In this chapter we introduce and survey some of the query languages developed for nested relational databases, and examine in detail the TQL language, which we co-designed to support advanced text-based applications. The TQL language will be used in the next chapter to compare queries on different schemas.

Over the past decade many query languages have been developed as part of nested relational database research projects. Many of these languages are included in a survey by Korth and Roth [74]. We illustrate several of these query languages using our hyperbase example and discuss how they have influenced the development of TQL. Figure 3.1 illustrates a tuple containing the hyperbase node shown in Figure 1.4; this tuple is part of an instance of the relation node in the hyperbase schema defined in Figure 2.7. Figure 3.2 illustrates three tuples, part of an instance of relation document in the same hyperbase schema.

Formal algebras and calculi have been developed for nested relational databases as well as a number of practical query languages. In Section 3.1 we introduce
The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means.

Although the subject of this paper is ostensibly the computable numbers, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth.

The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions and so forth to one another.

This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.
<table>
<thead>
<tr>
<th>docid</th>
<th>title</th>
<th>authors</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>On computable numbers, with an application to the Entscheidungsproblem</td>
<td>Alan Turing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Computing machinery and intelligence</td>
<td>Alan Turing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Preliminary discussion of the logical design of an electronic</td>
<td>Arthur Burks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>computing instrument</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Herman Goldstine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>John von Neumann</td>
</tr>
</tbody>
</table>

Figure 3.2: Three tuples in an instance of the document relation
a calculus for nested relations, and algebras for nested relations and for nested sequences of tuples. Abiteboul et al. [1] and Beeri [15] have considered the relative expressive power of different algebras and calculi for complex objects and nested relations.

SQL has become a de facto standard query language for relational database systems. Due to the prevalence of SQL in the database community, many of the languages proposed for post-relational databases supporting nested relations have been designed to have an SQL flavour. Three of these languages (SQL/NF [110], NF² [37, 103], and SQL/W [76]) are described in Section 3.2.

Object-oriented extensions to some nested relational query languages have already been proposed, for example X-SQL/NF [74]. Scholl and Schek observe “that the algebra for nested relations can be used almost without any change as an object algebra” and “therefore, query languages for object models ... are much closer to languages for complex objects or nested relations than to flat relational ones” [131]. We follow this approach by using TQL as the query language for both the physical data model of Atlas and our logical data model for complex objects.

In section 3.3 we describe TQL [137], a text-based language that is the data definition, data manipulation, and query language used by the Atlas information management system [117]. The Atlas system is a research prototype information system supporting advanced features such as the ability to support hierarchical data structures (including nested relations), structured data types, and reference pointers for linking database objects. None of these features are supported in standard relational systems. The Atlas system has also been designed to efficiently support text processing applications [118] and so TQL contains a text data type and a number of text operators. The limitations of TQL are also reviewed.

In the final section we summarize those aspects of TQL that distinguish it from other query languages and give a brief overview of its implementation in Atlas.
3.1 Formal nested languages

In the following brief survey of formal languages for nested relational databases, we introduce the main concepts involved in the formal algebras and one of the formal calculi as these form the basis for the SQL-based languages discussed in Section 3.2. Using the hyperbase example shown in the nested relational schema of Figure 2.7 we give examples of nested relational algebras in Section 3.1.2 and a nested relational calculus in Section 3.1.1. In Section 3.1.3 we give some examples of a nested sequences of tuples algebra using the schema of Figure 2.8.

In illustrating different formal query languages for nested relations and nested sequences of tuples we use three queries. The first query requires a selection condition on tuples in the nested relation link.

**Query 3.1** For each node give its identifier and the identifiers of all nodes linked to it via hyperlinks.

<table>
<thead>
<tr>
<th>nodeid</th>
<th>links</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodeto</td>
</tr>
<tr>
<td>1001</td>
<td>4000</td>
</tr>
<tr>
<td></td>
<td>9000</td>
</tr>
</tbody>
</table>

The next query involves restructuring the document relation so that it contains a nested relation of titles rather than a nested relation of authors.
Query 3.2  Restructure the document relation into a relation giving for each author a list of the titles of the documents they have written.

<table>
<thead>
<tr>
<th>firstname</th>
<th>surname</th>
<th>titles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alan</td>
<td>Turing</td>
<td>On computable numbers, with an application to the Entscheidungsproblem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computing machinery and intelligence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;</td>
</tr>
<tr>
<td>Arthur</td>
<td>Burks</td>
<td>Preliminary discussion of the logical design of an electronic computing instrument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;</td>
</tr>
<tr>
<td>Herman</td>
<td>Goldstine</td>
<td>Preliminary discussion of the logical design of an electronic computing instrument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;</td>
</tr>
<tr>
<td>John</td>
<td>von Neumann</td>
<td>Preliminary discussion of the logical design of an electronic computing instrument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;</td>
</tr>
</tbody>
</table>
The third query involves joining tuples in the nested relation `contents` in the `document` relation with tuples in the separate relation `node`.

**Query 3.3** *Find the title and contents of each document authored by Turing.*

<table>
<thead>
<tr>
<th>title</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>On computable numbers with an application to the Entscheidungsproblem</td>
<td>ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM by A.M. Turing [Received 28, May, 1936 — Read November, 1936] The “computable” numbers may be described briefly as the real numbers whose expressions as a decimal ... ... can be written down by a machine.</td>
</tr>
</tbody>
</table>

### 3.1.1 Nested calculus

The traditional tuple relational calculus needs to be extended to handle nested relations. The tuple variables in the tuple calculus for nested relations need to range over attributes which can be set valued as well as atomic valued. To allow tuple variables to range over set valued attributes it is necessary that equality comparison
of relations (including set constants) is allowed and that attributes can be equated with tuple calculus queries.

In the tuple calculus for nested relations, Query 3.1 can be expressed as follows, where the tuple variable \( u \) ranges over the tuples in the relation node and the tuple variable \( v \) ranges over the tuples in the nested relation links, and where the resultant tuple \( t \) has arity 2 and the resultant tuple \( s \) in the nested table has arity 1.

\[
\{ t^{(2)} \mid (\exists u) \ ( u \in \text{node} \\
\quad \land t[1] = u[1] \\
\quad \land t[2] = \{ s^{(1)} \mid (\exists v) \ ( v \in u[4] \\
\quad \quad \land s[1] = v[1] \\
\quad \quad \land v[2] = 'hyperlink' ) \} \}
\]

Query 3.2 can be expressed in the tuple calculus for nested relations as follows, where the tuple variables \( u \) and \( w \) range over the tuples in the relation document, and the tuple variables \( v \) and \( x \) range over the tuples in the nested relation authors.

\[
\{ t^{(3)} \mid (\exists u) \ ( u \in \text{document} \\
\quad \land v \in u[3] \\
\quad \land t[1] = v[1] \\
\quad \land t[2] = v[2] \\
\quad \land t[3] = \{ s^{(1)} \mid (\exists w) \ ( w \in \text{document} \\
\quad \quad \land s[1] = w[2] \\
\quad \quad \land (\exists x) \ ( x \in w[3] \\
\quad \quad \quad \land x[1] = t[1] \\
\quad \quad \quad \land x[2] = t[2] ) \} \}
\]

Finally, Query 3.3 can be expressed in the tuple calculus for nested relations as follows, where the tuple variables \( u \) and \( w \) range over the tuples in the relations document and node, and the tuple variables \( v \) and \( x \) range over the tuples in the
3.1.2 Nested algebras

In a nested relational algebra the basic relational algebra operators — union(∪), set difference(−), Cartesian product(×), projection(π), and selection(σ) — can behave as in the standard relational algebra except that the domains of these operators may be set-valued as well as atomic-valued.

To handle nested relations the traditional relational algebra is also extended with the addition of two operators called nest(ν) and unnest(µ), as described, for example, by Fischer and Thomas [51]. The nest operator takes a relation, groups the tuples with the same values for the unspecified attributes, and creates a nested relation using the specified attributes. The unnest takes a nested relation and does the reverse by flattening it. The nest and unnest operators are not inverse functions unless the database is maintained in partitioned normal form.

In Query 3.1, we unnest node, select the tuples with lkind equal to 'hyperlink', project the attributes nodeid and nodeto, and nest the result on nodeto forming the nested relation links.

\[ \nu \text{links} = (\text{nodeid}, \text{nodeto})\left( \pi (\text{nodeid}, \text{nodeto})\left( \sigma (lkind=\text{'hyperlink'})\left( \mu \text{links} \text{node} \right) \right) \right) \]

The following nested algebra expression represents Query 3.2.

\[ \nu \text{titles} = (\text{title})\left( \pi (\text{firstname}, \text{surname}, \text{title})\left( \mu \text{authors} \text{document} \right) \right) \]

Query 3.3 can be expressed in the nested algebra by unnesting both document and
The nested relational algebra that contains the operators $\cup$, $-$, $\times$, $\pi$, $\sigma$, $\nu$, and $\mu$ is a minimal relational algebra of equivalent power to the nested relational calculus restricted to safe expressions[111]. A safe expression is one in which the answer can be computed in finite time by examining only the relations and constants mentioned in the expression.

Other algebra operators have been proposed. For example, Roth et al. [111] describe extended union($\cup^e$), extended intersection($\cap^e$), extended difference($-^e$), extended natural join($\bowtie^e$), and extended projection($\pi^e$) operators. Use of these extended operators maintains nested relational databases in partitioned normal form. Null values present particular problems for nested relational algebra operations and these have been considered by Roth et al. [112]. They show how to overcome these problems using three kinds of null values, a no-information null, an unknown null, and a non-existent null.

Schek and Scholl [123] describe another extension to the nested algebra that recursively allows algebra expressions as attributes in projection and other operators. These nested expressions have been used in the SQL-based languages discussed in Section 3.2. In Query 3.1 we project node over the atomic attribute nodeid as well as the nested relation of nodeids which is the result of a query on the nested relation links.

Another example of the recursively defined algebra is the following expression for Query 3.3.

$$\pi_{\text{nodeid},(\pi_{\text{node}}(\sigma(\text{lkind} = \text{'hyperlink'}))\text{links})}\text{node}$$
Other variants on these algebras have been described by, for example, Abiteboul and Bidoit [2], Orman [98], and Ozsoyoglu et al. [99].

### 3.1.3 Nested sequences of tuples algebra

Güting and Zicari [57] and Güting et al. [58] describe a many-sorted algebra for nested sequences of tuples (NST) data model described in Section 2.4.2. The main features of the NST algebra are that it is many-sorted rather than one-sorted, and that the lambda operator (\(\lambda\)) has been introduced for restructuring hierarchical objects. The sorts of the NST algebra are numbers, text, Boolean, and complex objects. The NST algebra also handles null values.

The syntax of the NST algebra uses a postfix notation wherever any of the arguments may be complex objects. The lambda operator “allows newly constructed objects to be embedded into existing composite objects” [58]. For example, in Query 3.1 the complex object `links` is replaced by a new complex objects `links` containing a single attribute `nodeto` and only those tuples concerning hyperlinks.

\[
\text{node } \lambda[\text{links}:\text{links} \sigma[\text{lkind} = 'hyperlink'] \pi[\text{nodeto}]] \\
\pi[\text{nodeid, links}]
\]

The lambda operator is the most distinctive and versatile operator for manipulating complex objects in the NST algebra, however, other operators also manipulate complex objects. Query 3.2 makes use of the distribute (\(\delta\)), grouping (\(\gamma\)), and unpack (\(\mu\)) operators as follows.

\[
\text{document } \delta[\text{authors, titles} \pi[\text{authors}]\mu \gamma[\text{firstname, surname} \{\text{titles}\}]
\]

The distribute (\(\delta\)) operator is similar but more flexible than the the nested relational unnest operator because it distributes selective attributes “down” into a nested sequence. The grouping operator (\(\gamma\)) is similar to nest in the nested relational algebra, and the unpack (\(\mu\)) operator removes a level of nesting Query 3.3 is expressed in the
NST algebra as follows.

\[
\begin{align*}
\text{document} & \quad \sigma[\text{authors EXISTS\{surname = 'Turing'}}]\] \\
& \quad \lambda[\text{contents : contents node \bowtie [contents.nodeid = node.nodeid]}] \\
& \quad \pi[\text{data}] \\
& \quad \pi[\text{title, contents}]
\end{align*}
\]

The NST algebra is more complicated than the nested relational and recursively defined algebras described in the previous section; however, it may provide a better formal basis for document database systems because of the advantages of the NST data model in supporting ordering. The query language TQL has adopted a similar many-sorted approach to the NST algebra and also supports ordering while aiming to providing a user friendly query language.

### 3.2 SQL-based languages

In this section we discuss three SQL-based query languages which have influenced the design of TQL: SQL/NF provides the core syntax for TQL, NF\(^2\) includes the nested lists as in the NST model, and SQL/W provides the concept of reference attributes and implicit joins. The development of SQL3 [65] means that within a few years SQL systems will support collections of objects [14] in a similar way to these languages.

#### 3.2.1 SQL/NF

SQL/NF was developed by Roth et al. [109] as an extension to SQL to handle nested relations. In the process of extending SQL a number of improvements were made to make the language more orthogonal, that is “separate treatment for distinct concepts, and similar treatment for similar concepts” [109]. TQL, which we describe in detail in Section 3.3, closely follows the syntax of SQL/NF.

An SQL/NF schema for the hyperbase example is shown in Figure 3.3. In SQL/NF, a select-from-where (SFW) expression is allowed anywhere in a query that a relation could appear, for example, in the following expression for Query 3.1.
Figure 3.3: Declaration in SQL/NF for hyperbase
SELECT nodeid,
    (SELECT nodeto
    FROM   links
    WHERE  lkind = 'hyperlink')
FROM node.

The nested algebra operations nest and unnest allow arbitrary restructuring. These operators can be used with an aliasing feature (AS) to rename attributes, for example, in the following expression for Query 3.2.

NEST (SELECT firstname, surname, title
    FROM (UNNEST document ON authors))
ON title AS titles.

The aliasing feature can also simplify some queries with repeated subqueries.

Joins can be performed on nested tables as in Query 3.3.

SELECT title,
    (SELECT data
    FROM   contents, node
    WHERE  contents.nodeid = node.nodeid)
FROM   document
WHERE  EXISTS (SELECT *
    FROM   authors
    WHERE  surname = 'Turing')

Other advantages of SQL/NF are that functions are applied to relations rather than attributes, and the elimination of the GROUP BY and HAVING clauses of SQL. These features of SQL/NF have been incorporated in TQL.

3.2.2 \( \text{NF}^2 \)

The extended \( \text{NF}^2 \) system described in [105] provides a data model and algebra supporting tables that can be either ordered (sequences of tuples as in the NST
model described in Section 2.4.2) or unordered (relations, that is, sets of tuples). These lists or sets are not restricted to having elements of type tuple but may be lists or sets of scalars (atomic data types such as numeric, character or Boolean) or may be lists or sets of tables (either ordered or unordered). These are included in the language in an orthogonal way so that the basic SFW expression is a generic constructor for both sets and lists.

Sets are represented by expressions of the form \{...\}, lists by expressions of the form <...>, and tuples by expressions of the form <|$\ldots$|$. To create the tables in the hyperbase example the NF$^2$ declaration in Figure 3.4 would be required. In NF$^2$, Query 3.1 is expressed as follows.

```sql
SELECT <$\ldots$| nodeid: u.nodeid,

    links: (SELECT nodeto: v.nodeto

        FROM v IN u.links

        WHERE v.lkind = 'hyperlink') |>

FROM u IN node
```

In NF$^2$, Query 3.2 is expressed as follows.

```sql
SELECT <$\ldots$| firstname: x.firstname,

    surname: x.surname,

    titles: (SELECT title: u.title

        FROM u IN document,

        v IN u.authors

        WHERE v.firstname = x.firstname

        AND v.surname = x.surname) |>

FROM w IN document

x IN w.authors
```
CREATE OBJECT document {
  <| docid: INTEGER,
      title: CHAR,
      authors: <
        <| firstname: CHAR,
            surname: CHAR
       |> >
  |
} contents: <
  <| nodeid: INTEGER,
      |
  |
}> |

CREATE OBJECT node {
  <| nodeid: INTEGER,
      ofdoc: INTEGER,
      data: CHAR,
      links: <
        <| nodeto: INTEGER,
           lkind: CHAR,
           display: CHAR
        |
    |> >
  |
}"Figure 3.4: Declaration in NF² for hypertext database
In NF², Query 3.3 is expressed as follows.

```
SELECT <| title: u.title,
    contents: SELECT data: x.data
            FROM x IN node
            WHERE x.nodeid = v.nodeid) |>
FROM u IN document,
    v IN u.contents,
    w IN u.authors
WHERE EXISTS (w IN u.authors: w.surname = 'Turing')
```

Amongst possible further extensions to NF² described by Pistor and Traum-mueller [105] is the provision of a reference data type. This was included the query language SQL/W, which we describe next. Another extension is the support for recursive queries developed by Linnemann [79].

### 3.2.3 SQL/W

SQL/W was designed by Larson as the query language for LauRel [76], a prototype database system being developed at the University of Waterloo. TQL includes the concept of references and implicit joins from SQL/W [76] and also the concept of a structured tuple type.

An SQL/W schema for the hyperbase example is shown in Figure 3.5, and the Queries 3.1, 3.2, and 3.3 on this schema can be written as follows.

**Query 3.1**

```
SELECT nodeid,
    (SELECT nodeto
     FROM links
     WHERE lkind = 'hyperlink')
FROM node;
```
CREATE DATABASE hyperbase

  owner is dbadmin;

CREATE RELATION document OF hyperbase

  docid NUMERIC(5), KEY;
  title STRING(80);
  authors SETOF (
    firstname STRING(20);
    surname STRING(20);
  )
  CONTENTS SETOF REF node;

CREATE RELATION node OF hyperbase

  nodeid NUMERIC(8);
  ofdoc REF document;
  data STRING;
  links SETOF (
    nodeto REF node;
    lkind STRING(9),
    VALUES ('next', 'previous', 'parent', 'hypertext');
    display STRING;
  );

Figure 3.5: Declaration in SQL/W for hyperbase
Query 3.2

```
GROUP (
    SELECT firstname, surname, title
    FROM (UNNEST document ON authors)
)
ON firstname, surname FORMING titles;
```

Query 3.3

```
SELECT title,
    (SELECT hasnode.data
     FROM contents)
FROM document
WHERE 'Turing' IN (authors[surname]);
```

### 3.3 TQL

TQL was implemented by Alan Kent as the query language for the Atlas database system [117]. TQL provides the ability to define schemas in the physical data model of Atlas described in Section 2.5.1, and to query and update data stored in instances of these schemas. Like many other post-relational query languages, TQL has an SQL flavour but is not a strict extension of SQL. TQL closely follows the syntax of SQL/W; for example, Queries 3.1, 3.2, and 3.3 are the same in TQL as in SQL/W. A major difference between TQL and the other nested relational query languages is the support for applications using free text retrieval. In TQL text is treated as an atomic type and a number of operations on that type are supported.

#### 3.3.1 Select–from–where

As in SQL, the basic construct of a query in TQL is a `SELECT-FROM-WHERE` (SFW) expression. An SFW expression specifies a selection constraint (the `WHERE` clause)
which restricts which tuples in the table or tables (in the \texttt{FROM} clause) are retrieved in the result, and a projection list (the \texttt{SELECT} clause) of attributes or expressions based on attributes, to project from each tuple retrieved. An SFW expression can also be used to define joins. Note that the list of expressions for projection can return atomic values or tuples or nested tables.

**Query 3.4** *List the authors of the paper entitled ‘Intelligent machinery’.*

\[
\text{SELECT authors} \\
\text{FROM document} \\
\text{WHERE title = 'Intelligent machinery'};
\]

TQL also supports a functional notation for the representation of SFW queries where the table on the \texttt{FROM} clause is specified first, followed by an optional \texttt{select} clause in square brackets. Functional notation has been advocated in a number of query languages based on object oriented data models [62]. Using the functional notation the Query 3.4 could be expressed as

\[
\text{document[authors] WHERE title = 'Intelligent machinery'};
\]

It is sometime necessary to define an alias for a table, tuple or attribute identifier. Examples of where this is necessary include the requirement to disambiguate references to variables (when, for example a table is joined to itself) and the requirement to name attributes or tables in those cases where no identifier exists. An alias only lasts for the current query. New tuples attributes can be constructed in the \texttt{SELECT} clause by using ‘(‘ and ‘)’.

**Query 3.5** *List each document’s identifier, title, and authors in a tuple structure called ‘header’ which is separate from the contents in an attribute called ‘parts’.*

\[
\text{SELECT (docid, title, authors) AS header,} \\
\text{contents AS parts} \\
\text{FROM document};
\]
In order to support nested tables, TQL has a recursively defined syntax so that SFW expressions can contain nested SFW expressions. As such, it allows an expression that returns a table to be used within a query at any place that a table name can be used. This property of TQL, which follows the principle of orthogonality, allows for the support of an incremental approach to the formulation of complex queries and also simplifies query formulation in many cases. In TQL it is possible to use a query expression in a `SELECT` or `FROM` clause as well as in a `WHERE` clause. Other examples are provided by Roth et al. [110] to illustrate the importance of supporting an orthogonal syntax and the advantages of a more recursively defined SQL.

### 3.3.2 Explicit and implicit joins

The recursive syntax allows expressions to be nested to depth within a query, this supports a table being joined at any level within a hierarchical structure. The following query joins the nested table `contents` in the outer-level table `document` with the outer-level table `node`.

**Query 3.6** List the contents of the paper entitled ‘Intelligent machinery’.

```sql
SELECT (SELECT data
    FROM   contents, node
    WHERE  contents.hasnode = node.nodeid)
FROM   document
WHERE  title = 'Intelligent machinery';
```

Reference attributes allow implicit joins between tuples in a table containing the reference and tuples in the referenced table. Each of the attributes in the referenced tuple is directly available (through the implicit join) when querying on the tuple containing the reference. Using an implicit join, Query 3.6 could be expressed as follows.
SELECT (SELECT hasnode.data FROM contents)
FROM document
WHERE title = 'Intelligent machinery';

If there is no name clash it is not necessary to specify the reference attribute name. It is also possible to combine the functional and SFW notations by using linkrefs[lkind] as the functional notation SELECT lkind FROM linkrefs. Thus the query could be expressed as follows.

```
SELECT contents[data]
FROM document
WHERE title = 'Intelligent machinery';
```

Unpacking attributes which are tuples is straightforward; for example, the following query unpacks the tuple structure display. Unnesting or unfolding attributes which are tables is more complex and is dealt with in Section 3.3.4.

**Query 3.7** *For each document list the title and authors.*

```
SELECT title, authors[author.*]
FROM document;
```

The * unpacks all attributes of display and is equivalent to the following expression.

```
SELECT title,
    authors[author.firstname, author.surname]
FROM document;
```

This query can be written even more simply as follows.

```
SELECT title, authors[firstname, surname]
FROM document;
```

If all tuples from the document table are required to be kept (even if the join condition fails) then it is necessary to perform an outer join; this can be specified either by adding a `PRESERVE` clause\(^1\) to the end of the SFW expression or by using an implicit join, since implicit joins are always outer joins.

---

\(^1\)PRESERVE is not implemented in Atlas 1.8.
3.3.3 Support for text attributes

Since Atlas is designed for applications supporting free text, a number of text operators are supported by TQL. These operators support searching of free text on a word-by-word basis and the transformations of text to soundex, stemmed, and case converted forms. In TQL, text is treated as an atomic type.

TQL supports the searching of free text using the \texttt{CONTAINS} operator to specify that combinations of words have to appear within a string of text. The \texttt{CONTAINS} operator returns true if a text field holds a word or word phrase. A phrase is built from a list of words that must be adjacent in the text field. Phrases can be combined with ‘&’ (and) or ‘|’ (or). In a free text expression, the control over which method of comparison should be used can be done word by word.

\textbf{Query 3.8} Find all nodes containing the phrases 'computable numbers' or 'computable functions'.

\begin{verbatim}
SELECT nodeid
FROM node
WHERE data CONTAINS ('computable' 'numbers') |
     ('computable' 'functions');
\end{verbatim}

There are a number of ways a word in a query can be compared to a word of text. For example, it is often useful to compare the stems of words so that a query containing 'privacy' would match a document containing 'private' or 'privation'. Another form of comparison is based on the use of the soundex algorithm and results in matches if a query term sounds the same as a document term. Exact matching and case insensitive matching are also supported. For free text queries, the default type of comparison used is that which is defined in the database schema — described in section 3.3.6. Typically, stemming is used for free text, while soundex is often required for names of people or places. Although the default form of comparison is defined in the schema, it is possible to specify a query in which the default is overridden. For example stemming algorithms sometimes result in unrelated words reducing to the same stem. It may therefore be necessary to specify
exact matching in some free text or string expressions. To achieve this, TQL provides single prefix characters (transformation marks) to select the method of comparison. The possible transformations are: exact match (!), ignore case (^), stemming (~), and soundex (@). There is no transformation in TQL for specifying the synonymous comparison. In Atlas, stemming is implemented using Lovins’s algorithm [82] and soundex uses Russell-Soundex algorithm as described by Knuth [73].

The absence of a prefix transformation indicates that the default comparison is to be used. It is also possible to specify indexing using these transformations; this is described in section 3.3.6.

Any of the transformations can be applied to the equality and inequality operators. Thus ! = means the two text operands match exactly (that is, they are identical), ^ = means the two text operands are equal if case is ignored, ~ = means the stems of the two text operands are the same, and @ = means the two text operands sound the same under soundex.’ Similarly for inequality operators ! <> , ^ <> , ~ <>, and @ <>

**Query 3.9** Find all nodes which contain words sounding like ’organization’;

```sql
SELECT nodeid
FROM node
WHERE data CONTAINS @ 'organization';
```

The specification of substrings in queries on text fields is supported (as in SQL) using the LIKE construct. The LIKE operator uses % and _ for pattern matching as in SQL — % matches zero or more characters and _ matches a single character. Only the ! and ^ transformations can be applied to the LIKE operator.

**Query 3.10** Find all nodes which contain words beginning with ’organ’.

```sql
SELECT nodeid
FROM node
WHERE data CONTAINS ^ LIKE 'organ%';
```
If many text documents are returned by a query containing a disjunction it is helpful if results returning are ordered by their likely relevance to the original query. The query mechanisms of most information retrieval systems rank documents in order of their relevance to the user query, using the inner product document weighting function [122]. In TQL this can be achieved with the rank command as in Query 3.11.

**Query 3.11** Rank all nodes which containing either or both of the words beginning `relation` and `data`.

```sql
SELECT nodeid
FROM RANK node
WHERE data CONTAINS (^ LIKE 'relation%') | (^ LIKE 'data%');
```

### 3.3.4 Table and tuple operators

The equality and inequality operators (= and <>) can be used on tables and tuples as well as atomic values. However, relative operators (<, <=, >, >=) only apply to atomic attributes. Two tuples are equal if they are type compatible and the pairs of values for each attribute in the tuples are equal. Two tables are equal if they are type compatible and each table has the same tuples in the same order.

**Quantifiers and set comparison**

`EXISTS` returns true if the table subexpression contains at least one tuple, false otherwise. There are two other forms which are automatically mapped to `EXISTS` expressions. `ANY expr WHERE cond` returns true if any tuple in the table subexpression matches the `WHERE` condition and `ALL expr WHERE cond` returns true if all the tuples in the table subexpression match the condition.

`SUBSET` and `SUPERSET` take two tables of identical structure and return true if all the tuples in one table exist in the other.
The **IN** operator returns true if the left atomic or tuple subexpression is in the set of values returned by the right table subexpression, for example:

**Query 3.12** *Find all nodes which contain hyperlinks to other documents.*

```
SELECT nodeid
FROM node
WHERE 'hyperlink' IN (links[lkind]);
```

The **HAS** operator is identical to **IN** except that the operands are reversed and different types of comparison (including text transformations) can be used.

**Aggregate functions**

In TQL, the argument to an aggregate function such as **SUM**, **AVG**, **MAX**, or **MIN** is a SFW-expression that returns a vector (a table with one attribute) of numeric values. The function **COUNT** takes as an argument any table. This differs from SQL/NF which also allows table arguments to **SUM**. Like SQL/NF, TQL was designed to incorporate an orthogonal implementation of functions. SQL, on the other hand, is weak in this respect [41]. As Date has observed [40], in SQL, the argument to a function is specified in a most unorthodox manner and as a consequence function references can only appear in a small set of special-case situations.

**Finding a row in a table**

**ROWNUM** returns the tuple number of the current tuple. The current tuple is the tuple retrieved on the **FROM** clause of a SFW expression. **ROWNUMs** for nested tables are sequenced 1, 2, 3, and so on. For outer level tables however, **ROWNUM** is the actual record number on the file. Thus if a record is deleted, the **ROWNUMs** will no longer be contiguous.

**Query 3.13** *Show the kind of link for the first listed link for each node in paper entitled Intelligent machinery.*
SELECT nodeid, (links[lkind] WHERE ROWNUM = 1)
FROM node
WHERE title = 'Intelligent machinery';

ROWNUM cannot be applied to all table expressions, it can be applied to outer level tables (GLOBAL, LOCAL, or TEMP) and nested tables but not tables generated as intermediate results in a query. If the row does not exist ROWNUM returns NULL.

Operations on tables

A number of the relational algebra operators (UNION, INTERSECT, MINUS, TIMES, and JOIN) are provided as abbreviations for SFW expressions. In addition TQL provides three special operators (UNNEST, UNFOLD, and GROUP) to restructure tables.

The UNNEST operator is used to flatten the result of a table subexpression which contains a nested table. For each tuple in the nested table being unnested on, a copy of all the other attributes is made. If the nested table does not contain any tuples, then a single tuple is returned with the nested table attributes being given null values. UNFOLD is similar to UNNEST but when an empty subtable is unfolded the corresponding tuples are removed. The following two queries illustrate the difference between the UNNEST and UNFOLD operators.

Query 3.14 Produce a 1NF table of document and node identifiers.

SELECT docid, nodeid
FROM (UNNEST document ON contents);

Query 3.15 Produce a 1NF table of document and node identifiers without retaining any documents that have no nodes.

SELECT docid, nodeid
FROM (UNFOLD document ON contents);

The operator ':' is used as an abbreviation\(^2\) for UNFOLD ... ON, for example query 3.15 could also be expressed as follows.

\(^2\)This is used instead of the automatic unfold of LauRel [76] because in many situations an automatic unfold becomes difficult to understand.
SELECT docid, nodeid
FROM document:contents;

The GROUP operator builds a nested table by grouping tuples with common values in unnested attributes together to form a single new tuple.

**Query 3.16** Restructure the document table to list documents published by each set of authors; where there are duplicate sets of authors regroup the authors table to form a list of titles of documents published by that group.

GROUP document
ON authors
FORMING docs_by_auth;

The result of this query is a complex object of the form

```plaintext
document [
  authors [
    name(firstname TEXT,
      surname TEXT)
  ]
  docs_by_auth [
    docid INTEGER,
    title TEXT,
    contents TEXT
  ]
]
```

As in SQL, duplicates are not removed in an ordinary TQL query. The function DISTINCT takes a table as an argument and removes duplicates. Two tuples are considered to be duplicates if all atomic values are equal and any nested tables contain the same set of tuples where the ordering of tuples is significant. The table returned has the same structure as the subexpression but may contain fewer tuples. The ordering of tuples in the returned table is not guaranteed. Another similar way to express query 3.16 which does not require the GROUP operator is as follows.
DISTINCT (SELECT authors,  
    (SELECT * BUT authors  
    FROM document AS docs_by_auths  
    WHERE doc.authors = docs_by_auths.authors)  
    FROM document AS doc);

### 3.3.5 TQL data manipulation language

The DML command **INSERT** is used to add new tuples to a table or when used with **UPDATE** can add new tuples to a nested table. There are two forms of the **INSERT** command — **INSERT VALUES** and **INSERT :=**. The first form inserts tuples. The second form wipes the table first before inserting the new tuples. That is, it replaces the contents of the table with a new set of tuples. The following query adds a tuple to the **node** table using the **INSERT** command.

#### Query 3.17

```sql
INSERT node VALUES [
    101234,
    101,
    'Some remarks on the Privacy Act 1988 ... ',
    [(101233),'previous'('lower left button (Previous)')]
];
```

The result of any query can be used as data for insertion, not just constant tuples. Also, a list of attribute names can be specified allowing the attribute names to be ordered as desired within the **INSERT** command. Any attributes that are not included are assigned the value **NULL**.

**DELETE** removes matching tuples from a table (or nested table). A **WHERE** clause is used to restrict which tuples to delete. A **DELETE** command can be used on nested tables by nesting the command in an **UPDATE**.

The **UPDATE** command is used to update tuples in a table. Update commands can be used to assign a new value to an atomic field, or modify nested tables using
a nested DML command. Thus an update command can contain nested \texttt{INSERT}, \\
\texttt{DELETE} or \texttt{UPDATE} commands. The set clauses are evaluated in the order of appearance.

\textbf{Query 3.18} \textit{Add the node 1234 to the Privacy Act 1988.}

\begin{verbatim}
UPDATE document
SET   (INSERT contents VALUES [1234])
WHERE document.title = 'Privacy Act 1988'
\end{verbatim}

Since an update is potentially modifying a number of values within the database it should be implemented as an atomic transaction.

\section*{3.3.6 TQL data definition language}

TQL supports three types of outer level tables — \textit{global}, \textit{local}, and \textit{temporary}. Global tables reside in a shared database and are accessible by all users who connect to that database. All shared tables must be global and only global tables can be indexed.

Local tables are local to each user. One user cannot access a local table belonging to another user. Local tables do not have indexing and are intended for local storage space. Local tables are independent of the database that the user connects to and so can be used to transfer data between databases. Local tables are also useful for applications for which it is required to keep user dependent parameters in a database table.

Temporary tables are local to a session and are deleted at the end of a session.

In order to create an outer level table using TQL, it is necessary to modify the current \textit{schema definition}. As in SQL, TQL contains commands both to create and to drop tables. The table creation command can provide formatting information for the new table. A separate command is used to create an index for a table.

TQL also supports query-only views which provide a powerful mechanism for the support of high level user abstractions of the data. Views are also a useful mechanism
for complex query formulation. Once created, a view can then be treated as a normal table in queries.

Creating and deleting tables

The following table creation command creates the node table including defaults for formatting and text transformations.

```
CREATE GLOBAL TABLE node [
    nodeid INTEGER,
    ofdoc REF document,
    data ~TEXT FORMAT 'width=80',
    links [
        nodeto REF node,
        lkind TEXT,
        display TEXT
    ]
] KEY = (display)
] KEY = (nodeid);
```

Prefix characters are used to define the default type of equality matching for text fields. In the absence of prefix characters, exact matching is used as the default.

For all fields, it is also possible to specify a format for the output of queries in TQL. This is an important feature for displaying text fields on a terminal without the aid of a forms package. Formats are used by the TQL command interface to format the output of a query. TQL supports variable length fields which may be of infinite length. When displaying the result of a query however, a column width is needed to output the result of the query in a readable form. Columns can be assigned default formats when the table is defined by adding a FORMAT clause to one or more of the attributes defined in the CREATE command. Alternatively the FORMAT statement can be used in a query to override the default format for the result of an expression.
Tables can be deleted using the `DROP` command, the following example removes the global table `node` from the current database.

```sql
DROP GLOBAL TABLE node;
```

**Index creation**

To create an index on a global table, the `CREATE INDEX` command must be used. Currently Atlas supports three indexing schemes: superimposed coding schemes ([116](sic)), linear hashing (`linh`), and direct indexes for tables with a key attribute whose values are distinct small integers (`direct`). To index the contents of tables and these schemes require a number of parameters to be specified. In physical schema design we assume superimposed coding index schemes and we explain the parameters of such schemes in Chapter 6. To assist users, the `ANALYSE INDEX` command analyses a table of data, the index currently defined, and automatically selects appropriate parameters.

Consider the following command.

```sql
CREATE sic INDEX node [
    !nodeid,
    ^(data),
    ~WORDS(data)
];
```

This command will cause the fields `nodeid` and `data` to be indexed; the fields `ofdoc` and `links` are not indexed; and two indexes are constructed for the `data` field. Each attribute name is prefixed by a transformation character. This alters the actual value used for indexing. For example, `!nodeid` causes the exact value of the people number to be indexed on. Qualifying the field `data` by `WORDS` specifies that phrase indexing is performed on that field. By specifying both `^(data)` and `~WORDS(data)`, both individual words in `data` mapped to a single case and the stemmed version of phrases in `data` will be indexed.
Once the index has been created, data can be loaded into the table. The **ANALYSE INDEX** command can then be used to design the index parameters. The table must contain some data before the **ANALYSE INDEX** command can be used as it samples the data in the table to determine information such as the average number of index terms per record.

**ANALYSE INDEX node;**

**ANALYSE** performs an analysis of a global table and designs parameters for the index appropriately; it takes into account such parameters as the database capacity and frequency of terms. A table of common words is also formed by sampling data in the database. At the end of the analysis, an index rebuild is invoked with the new database parameters.

### 3.3.7 Atlas architecture for TQL

Sacks-Davis et al. [117] describe the architecture of the Atlas system. The diagram in Figure 3.6 shows the relationships between the major subsystems. (Some relationships, such as the interrogation of schema structures by various subsystems, are not presented.) The four boxes with dashed borders show how the subsystems—the application layer, Atlas kernel, and central schema manager—are grouped into UNIX processes. Each application program creates a new Atlas kernel process. Kernel processes use the central schema manager to hold shared data structures such as the schema and indexing details. These data structures are kept in shared memory for fast access.

When a TQL query is submitted to the system via the Application Programming Interface (API), either as an ad-hoc query or a query embedded in an application program, it is sent to the kernel for processing. At this stage the query is represented as an expression tree, not in the original text format. In the kernel, the driver module determines that it is a query rather than some other type of command (such as a schema definition) and passes the request to the Data Manipulation Language

---

3Currently only available for sic indexes.
Applications

CTQL
API
Communications

Atlas Kernel

Driver

DML Driver

Type Checker
Rule Based Optimizer
DML Evaluator

Table Manager

Data File Manager

Index Manager

Index Manager

SIC LINH Direct

UNIX File System

Figure 3.6: Atlas system architecture diagram
(DML) driver, which is then responsible for processing the query. The DML driver uses the type checking module to check that the query is legal. Identifiers are resolved during this phase, but otherwise the expression tree is not affected. The rule-based optimiser then manipulates the expression tree into a form that is more efficient to evaluate. Finally the optimised expression tree is submitted to the DML evaluator.

### 3.3.8 Limitations of TQL

There are several limitations to TQL. Many of these have been identified by Kent [68]. These problems relate to compromises made in both the syntax and the type system to achieve a simple to use interactive query language. Nested query languages such as TQL are not compatible with SQL and for anything but the simplest queries require different expressions.

Some problems with TQL syntax relate to its complexity: there are too many reserved words and operators. This stems from the calling of functions (such as aggregates) using reserved words each with its own syntax — a better approach would have been to use a standard syntax for all functions and then use the Atlas kernel to resolve any function name clashes. A similar problem exists with the transformation symbols (\(!, ^, \sim, \&\)) they are useful but difficult to extend for any new transformations (a shortage of symbols). Also, the \(!\) transformation is confusing for C programmers, with \(!=\) meaning exact match rather than not equal.

There are some problems with the type system; these problems arose because TQL was designed to keep interactive queries simple. For example, an object comprising a tuple with one column can be used anywhere an object with a single column is needed. Although this made interactive queries simple it created confusion concerning the structures returned for embedded queries. Another problem with the type system was the structure returned by Cartesian product, a multiset containing pairs of tuples. This was introduced to avoid naming problems where there was a conflict of attribute names but created problems for the query optimizer because \((A \times B) \times C\) does not return the same structure as \(A \times (B \times C)\).
An important limitation of TQL not discussed elsewhere is that, in the implementation of TQL, the query language is used in the Atlas database system to query the physical data model described in Section 2.5.1. We propose that TQL can also be used as a query language for the logical data model described in Section 2.5.2. The \texttt{LIST OF} and \texttt{SET OF} constructs in the logical schema can be queried in the same way as a nested table. Similarly where an object class is not one of the built-in classes it can be treated in the same way as a reference attribute. This results in a straightforward language for querying such logical schemas. These mappings are considered in more detail in Chapter 4.

### 3.4 Summary

We have described a language TQL for nested relations. The language provides support for non-first normal form relations including text attributes, tuples, nested tables, and implicit joins using references. Compared with other languages proposed for the nested relational model, TQL has powerful support for text.

Some problems of query optimization for TQL and the nested relational model are still to be investigated. Particular issues are the complexity of join operators for nested tables and the efficient support of operators such as nest and unnest.

In the next chapter we describe how ease of query expression can influence the design of the logical database schema. The TQL language is used to compare ease of query expression on different logical schemas.
Chapter 4

Schema design

Database design is concerned with both logical representation and physical structures that allow efficient retrieval. Data models based on complex objects have advantages at both the logical and physical levels; in this chapter we contend that there is a significant difference between the complex objects required at the logical level and the complex objects required at the physical level. Logical schema design of complex objects should give a representation of data which is close to the world being modelled. The ideal logical design allows queries to be expressed in as simple a manner as possible, each entity being modelled as an object which may contain other objects as components. The ideal physical schema, on the other hand, should allow queries to be executed as efficiently as possible. The selection of the ideal nesting of entities at the physical level is an optimization problem; the goal is to find the structure which allows the most efficient querying.

If different schemas are defined for logical and physical design, then it is necessary to translate between these schemas. Such a translation is only valid if the logical schema is translated into an equivalent physical schema. We discuss different levels of schemas and the issue of schema equivalence in Section 4.1.

In Section 4.2 we use the query language TQL, described in the previous chapter, to compare queries on equivalent schemas based on the relational, nested relational, and physical object models. These queries are compared, in Section 4.3, with an equivalent schema based on our logical data model with complex objects. This
logical data model can be derived using a straightforward translation from the E-R model.

Given one logical schema, there is a choice of equivalent physical schemas. Each physical schema can be described using the data definition facilities of TQL. In Section 4.4 we discuss alternative physical schemas, and we present a method for translating our logical schema into any of these physical schemas. Our method uses an extended syntax for TQL that includes derived attributes. We also present a framework for finding the most efficient physical schema given a particular mix of queries.

The final section summarizes the main issues concerning the logical and physical design of databases with complex objects.

### 4.1 Schema equivalence

No single database schema can describe all aspects of a complex real-world situation in a way amenable to computers. So, often we wish to use more than one schema and more than one data model. In 1971 the CODASYL Database Task Group [31] proposed a three level architecture for database schemas which has been adopted widely; in 1982 it was approved as part of an international standard [56]. In this architecture there are three levels of schemas: *physical, logical*, and *external*.

**physical level/schema**

The *physical (or internal) level* is described by a *physical (or internal) schema* of the physical data structures using a physical data model.

**logical level/schema**

The *logical (or conceptual) level* is described by a *logical (or conceptual) schema* of the community view of the database using a logical data model.
external level/schema

The external (or view) level is described by a series of external schemas (or views) for different users or applications using a logical data model.

The three level architecture provides both logical data independence and physical data independence. Logical data independence means that changes to the logical schema should have minimal or no effect on the existing external schemas. Physical data independence means that changes to the physical schema should have minimal or no effect on the existing logical or external schemas.

The goal of external schema design should be to design a schema that allows users and application programmers to express as easily as possible their queries (including queries which update the database).

The logical schema combines the various external schemas into a single schema by a process of schema integration [12]. A good logical schema should allow all reasonable external views to be easily expressed and capture the essential integrity constraints on the database. In relational databases this has involved the process of normalization described in Section 2.3.3. An alternative approach to normalization has been to use a semantic data model to define the logical model (which is often called the conceptual model in this context). For example, Batini et al. [11] describe a methodology for design of office database schemas using a semantic data model. They derive a semantic data model, that is an extension of the entity-relationship model, directly from office forms, which are documents with a particularly structured style.

The goal of physical design is different. A good physical design is one which allows queries and updates to be efficiently processed. In relational databases this may involve de-normalization, the joining together of normalized relations to improve efficiency [46].

The different goals of the logical and physical levels are best resolved by using different schemas expressed using different data models. This requires a schema expressed using one data model to be translated into an equivalent schema expressed using a different data model. For example, Elmasri and Navathe [46] describe a
database design process in which a logical data model is used during the initial design of logical and external schemas; these schemas are then translated into the physical data model of the database system being used. Such translations have been developed between schemas in many different models; examples include: Biller’s [17] description of an algorithm for translating a network schema to a relational schema; Elmasri and Navathe’s [46] description of algorithms for translating from an entity-relationship schema, or an extended entity-relationship schema, to any of the classical data models: hierarchical, network or relational; and Leung and Nijssen’s [77] description of an algorithm for translating NIAM conceptual schemas into normalized relational databases.

Any mechanism for translating one schema to another must be based on a concept of schema equivalence. Batini et al. [12], in their survey of methodologies for database schema integration, identify three types of schema equivalence in the literature: behavioural equivalence (described by Atzeni et al. [8]), mapping equivalence (for example, described by Rissanen [107] in the context of join dependencies), and transformation equivalence (described by Navathe and Gadgil [94]). In this chapter we use mapping equivalence, a transitive, reflexive and symmetric relationship between schemas defined as follows.

**mapping schema equivalence**

Two schemas $S_1$ and $S_2$ are *mapping equivalent* if there exists a one-to-one mapping from $\{s_1 \mid s_1 \in S_1\}$ to $\{s_2 \mid s_2 \in S_2\}$. The two schemas $S_1$ and $S_2$ do not have to be expressed using the same data model.

The constraints occurring in the schemas are crucial in determining whether a one-to-one mapping exists because the constraints determine the valid instances of a schema. The above definition only considers static constraints, unlike Biller [17] who also considers dynamic contraints.

In the following sections, we compare several mapping equivalent logical and physical schemas.
4.2 Comparing queries on different schemas

The ease with which queries can be expressed depends on the database schema against which they are posed. McGee has suggested “that there is a tradeoff between the complexity of a data structure, and the complexity of queries directed against it” [87]. In this section we consider queries expressed against several database schemas. These include a relational schema, several alternative nested relational schemas, and a schema supporting complex objects. All these schemas can be described using part or all of the implementation data model of Atlas.

Consider a document database containing hypertext nodes involving four sets of entities that we wish to model: documents, authors, nodes, and links. Since we are concerned with the equivalence of schemas we must define more completely the constraints as well as the structure. In addition to key and domain constraints, we assume the following constraints on this database: each document has one or more nodes and zero or more authors; each node belongs to exactly one document and has zero or more links to other nodes; every author has written at least one document; and links are bidirectional (that is, if there exists a link from node $A$ to node $B$ then there exists a link from node $B$ to node $A$).

4.2.1 Relational schemas

The relational model provides a uniform treatment of all attributes; this has some advantages in allowing any ad-hoc query to be expressed. Many conceptually simple queries can, however, be complex to express when on relations which represent parts of decomposed entities. Join queries, which reassemble these decomposed entities, occur frequently in normalized relational databases and in a large database can

---

1Some documents may have no attributable author, for example, an Act of Parliament or a set of instructions.

2Arbitrary ad-hoc queries can only be expressed if the query language is sufficiently powerful. Chandra and Harel [26] define such a language which is able to express all queries that are computable. In practice, less powerful query languages are suitable in most cases; Chandra [25] categorizes the expressive power of various query language constructs. Abiteboul et al. [1] consider these issues in context of complex objects and nested relations,
involve many relations; such queries are in general difficult to express and expensive to execute if the underlying data is stored in flat relations.

In the relational approach we require the four normalized relations (document, author, node, and link) shown in the schema in Figure 4.1; it only differs from the relational schema given in Figure 2.6 by the addition of explicit constraints and renaming of attribute domains in line with Atlas. The normalized schema contains various key constraints, domain constraints, and a template dependency. It is in Project-Join Normal Form, because there are no join dependencies that are not a logical consequence of the keys, but it is not in Domain-Key Normal Form because the template dependency is not a logical consequence of the domain and key dependencies. The relational approach to the design of databases can produce relations which have split the entities involved. In this case a document entity is split across four relations: document, author, node, and link. Many queries in relational systems, including the following query, require joins to reassemble entities and to follow relationships between entities.

**Query 4.1** Find the authors and the content of the document entitled ‘Computing machinery and intelligence’.

To answer this query, using either SQL or TQL, we require the following join on the relation schemas in Figure 4.1.

```sql
SELECT firstname, surname, data
FROM document, author, node
WHERE title = 'Computing machinery and intelligence'
  AND document.docid = author.docid
  AND document.docid = data.docid
```

The above relational query will display the information in each node, but it will not organize the information in an easy-to-read fashion. Nevertheless, in a hyperbase system, the SQL query could be embedded in a host programming language that would display the node in an easy-to-read fashion. Such a solution requires
**CHAPTER 4 — SCHEMA DESIGN**

**Figure 4.1: Relational database schema**

```
<table>
<thead>
<tr>
<th>table</th>
<th>attributes</th>
<th>key</th>
<th>foreign key constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>document</td>
<td>docid INTEGER, title TEXT</td>
<td>(docid)</td>
<td>docid FOREIGN KEY REF document</td>
</tr>
<tr>
<td>author</td>
<td>firstname TEXT, surname TEXT, docid INTEGER</td>
<td>(firstname, surname, docid)</td>
<td></td>
</tr>
<tr>
<td>node</td>
<td>nodeid TEXT, ofdoc TEXT, data STRING</td>
<td>(nodeid)</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td>nodefrom TEXT, nodeto TEXT, lkind TEXT, display TEXT</td>
<td>(nodefrom, display)</td>
<td></td>
</tr>
</tbody>
</table>

**Foreign key constraints**

```

```

**Template dependency**

```

<table>
<thead>
<tr>
<th>hypothesis</th>
<th>link [ nodefrom nodeto lkind display ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>Y1</td>
<td>W1</td>
</tr>
<tr>
<td>Z1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conclusion</th>
<th>-------------------------------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y1</td>
<td>X1</td>
</tr>
<tr>
<td>W2</td>
<td>Z2</td>
</tr>
</tbody>
</table>

```

Figure 4.1: Relational database schema
an interface between two different languages, which adds to the complexity of the system; this has been described by advocates of object-oriented database systems as an *impedance mismatch* \[52\]. Attempts to minimize the impedance mismatch problem in conventional databases has led to the development of fourth generation languages. Fourth generation languages provide a mechanism for generating applications with easy-to-use interfaces; these then contain embedded database queries. An alternative solution to displaying information in an easy-to-read fashion has been developed Toyama \[142\]; his solution involves an extension to the relational language QUEL that allows hierarchical structured tables to be output from a query.

### 4.2.2 Nested relational schemas

A well-designed nested relational schema can be easier to query than a flat relational schema. The main issue when designing a nested relational schema is deciding which entities to nest inside other entities. For example, the entity *document* (and respectively the entities *author*, *node*, and *link*) has been chosen as the outermost level of nesting in the schema in Figure 4.2 (and respectively the schemas in Figures 4.3, 4.4, and 4.5). Other nestings are also possible. Sometimes when nesting occurs some attributes are no longer required, for example, whenever *link* is nested inside *node*, the attribute *nodefrom* can be removed. In general, when a relation with a composite key that includes a foreign key, is nested inside the relation referenced by the foreign key, the attributes in the foreign key can be omitted from the nested relation.

If we use TQL and ignore ordering of tuples (that is, consider only mathematical nested relations based on sets rather than the nested sequences supported by Atlas) we could express each of the schemas in Figures 4.2, 4.3, 4.4, and 4.5 as a view on the schema in Figure 4.1. For example, the view giving the schema in Figure 4.2 could be expressed as follows.
CHAPTER 4 — SCHEMA DESIGN

```
document [
    docid INTEGER,
    title TEXT,
    authors [
        firstname TEXT,
        surname TEXT
    ] KEY = (firstname, surname),
    contents [
        nodeid INTEGER,
        data TEXT,
        links [
            nodeto INTEGER,
            lkind TEXT,
            display TEXT
        ] KEY = (display)
    ] KEY = (nodeid)
] KEY = (docid)
```

Foreign key constraints

```
document.contents.links
    nodeto FOREIGN KEY REF document.contents
```

Template dependency

```
document[docid ... contents[ nodeid data link[ nodeto ...]]]
```

<table>
<thead>
<tr>
<th>hypothesis</th>
<th>X1</th>
<th>V1</th>
<th>Y1</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>conclusion</th>
<th>Y1</th>
<th>V2</th>
<th>X1</th>
</tr>
</thead>
<tbody>
<tr>
<td>U2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 4.2: Database schema nested on documents
CHAPTER 4 — SCHEMA DESIGN

author [
    firstname TEXT,
surname TEXT,
documents [
    docid INTEGER,
title TEXT,
    contents [
        nodeid INTEGER,
data TEXT,
        links [
            nodeto INTEGER,
lkind TEXT,
display TEXT
        ] KEY = (display)
    ] KEY = (nodeid)
    ] KEY = (docid)
] KEY = (firstname, surname)

Foreign key constraint

    author.documents.contents.links
    nodeto FOREIGN KEY REF author.documents.contents

Cardinality constraint

    NOT EMPTY (author[documents])

Template dependencies

<table>
<thead>
<tr>
<th>author[ firstname surname documents[docid title contents[*]]]</th>
<th>hypothesis</th>
<th>V1</th>
<th>W1</th>
<th>X1</th>
<th>Y1</th>
<th>Z1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>V2</td>
<td>W2</td>
<td>X1</td>
<td>Y2</td>
<td>Z2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>conclusion</td>
<td>Y1=Y2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|                 |            | Z1=Z2
| author[ ... ]  |            |-----------------------------|
| documents[docid ... contents[nodeid data links[nodeto ...]]] | hypothesis | U1 | X1 | V1 | Y1 |
|                 |            |-----------------------------|
|                 | conclusion | U2 | Y1 | V2 | X1 |

Figure 4.3: Database schema nested on authors
CHAPTER 4 — SCHEMA DESIGN

node [
    document ( 
        docid INTEGER,
        title TEXT,
        authors [ 
            firstname TEXT,
            surname TEXT
        ] KEY = (firstname, surname)
    ),
    nodeid INTEGER,
    data TEXT,
    links [ 
        nodeto INTEGER,
        lkind TEXT,
        display TEXT
    ] KEY = (display)
] KEY = (nodeid)

Foreign key constraints

node.links
    nodeto FOREIGN KEY REF node

Template dependencies

node[ document[ docid title authors[*] ] nodeid ... ]

hypothesis
    X1  Y1  Z1  W1
    X1  Y2  Z2  W2

conclusion
    Y1=Y2  Z1=Z2

node[ documents[*] nodeid data links[ nodeto lkind display ]]

hypothesis
    U1  X1  V1  Y1  W1  Z1

conclusion
    U2  Y1  V2  X1  W2  Z2

Figure 4.4: Database schema nested on nodes
link [  
  nodefrom (  
    document (  
      docid INTEGER,  
      title TEXT,  
      authors [  
        firstname TEXT,  
        surname TEXT  
      ] KEY = (firstname, surname)  
    ),  
    nodeid INTEGER,  
    data TEXT  
  ),  
  nodeto (  
    document (  
      docid INTEGER,  
      title TEXT,  
      authors [  
        firstname TEXT,  
        surname TEXT  
      ] KEY = (firstname, surname)  
    ),  
    nodeid INTEGER,  
    data TEXT  
  ),  
  lkind TEXT,  
  display TEXT  
] KEY = (nodefrom.nodeid, nodeto.nodeid, display)

Template dependencies

<table>
<thead>
<tr>
<th>link[ nodefrom(<em>) nodeto(</em>) lkind display ]</th>
<th>hypothesis</th>
<th>X1</th>
<th>Y1</th>
<th>W1</th>
<th>Z1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>conclusion</td>
<td>Y1</td>
<td>X1</td>
<td>W2</td>
<td>Z2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link[ nodefrom( document(docid title authors[*]) nodeid data) ... ]</th>
<th>hypothesis</th>
<th>X1</th>
<th>Y1</th>
<th>Z1</th>
<th>W1</th>
<th>V1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X1</td>
<td>Y2</td>
<td>Z2</td>
<td>W2</td>
<td>V2</td>
</tr>
<tr>
<td></td>
<td>conclusion</td>
<td>Y1=Y2</td>
<td>Z1=Z2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link[ nodefrom( document(*) nodeid data) ... ]</th>
<th>hypothesis</th>
<th>X1</th>
<th>Y1</th>
<th>Z1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X2</td>
<td>Y1</td>
<td>Z2</td>
</tr>
<tr>
<td></td>
<td>conclusion</td>
<td>X1=X2</td>
<td>Z1=Z2</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.5: Database schema nested on links
document_view :=
    SELECT docid,
    title,
    (SELECT firstname, surname
     FROM authors
     WHERE docid = document.docid) AS authors,
    (SELECT nodeid,
     data,
     (SELECT nodeto, lkind, display
      FROM links
      WHERE nodefrom = node.nodeid) AS links,
     FROM node
     WHERE docid = document.docid) AS nodes
    FROM document;

From the schemas in Figures 4.2 and 4.4 the original relational schema in Figure 4.1 can be reconstructed using a view. For example, using the schema in Figure 4.2 we can give the inverse of the previous view definition as follows.

document_view :=
    SELECT docid, title
    FROM document;

author_view :=
    SELECT firstname, surname, docid
    FROM (UNFOLD document ON authors);

node_view :=
    SELECT nodeid, docid, nkind, data
    FROM (UNFOLD document ON nodes);

link_view :=
    SELECT nodeid AS nodefrom, nodeto, lkind, display
    FROM (UNFOLD (UNFOLD document ON nodes) ON links);

If the integrity constraints are enforced on both schemas, in Figures 4.1 and 4.2,
then the views define a one-to-one mapping between the two schemas and hence these two schemas are equivalent. Similarly, the schemas in Figures 4.1 and 4.4 are equivalent. Such equivalence does not apply between the schema in Figure 4.1 and either schema in Figures 4.3 or 4.5. When there are cycles in the original schema it is unavoidable to nest some entities in more than one entity.

In order to compare ease of expression of different queries in a precise way, we use the following definition of simpler query.

**simpler query**

Given two TQL expressions $E_1$ and $E_2$ for the same query, we say that $E_1$ is *simpler* than $E_2$ if it contains less tokens, where a *token* is a TQL reserved word, operator, identifier, or constant. Likewise, we say that $E_1$ is *more complex* than $E_2$ if it contains more tokens.

Consider the following four TQL expressions of Query 4.1 using each of the nested relational schemas. The simplest expression of the query uses the schema in Figure 4.2 as follows.

```sql
SELECT authors, contents[data]
FROM document
WHERE title = 'Computing machinery and intelligence';
```

The following nearly as simple expression of the query uses the schema in Figure 4.4.

```sql
SELECT document.authors, data
FROM node
WHERE document.title = 'Computing machinery and intelligence';
```

Both the above queries are simpler than the join query required for the schema in Figure 4.1.

Using the schema in Figure 4.5 the following more complex query expression is required.
DISTINCT ( 
    SELECT authors, nodefrom(data) 
    FROM link 
    WHERE nodefrom.document(title) = 
        'Computing machinery and intelligence'; 
) GROUP ON authors FORMING contents

Use of the schema in Figure 4.3 requires an even more complex query expression.

DISTINCT ( 
    SELECT firstname, 
        surname, 
        (SELECT contents[data] 
            FROM documents 
            WHERE title = 
                'Computing machinery and intelligence') 
    FROM author 
    WHERE EXISTS 
        (documents 
            WHERE title = 'Computing machinery and intelligence')
);

The query on the schema in Figure 4.2 could be considered the simplest as it reflects the structure of the result required (though different users may find different expressions of the same query easier to understand).

In a situation where there is a range of ad hoc queries on the database, no one structure will adequately serve all query types. An illustration of a query which is expressed more simply on a different schema is Query 4.2.

**Query 4.2** Find the titles of all documents containing nodes linked to document whose title is ‘Computing machinery and intelligence’.

Using the schema in Figure 4.5 requires the following query expression.
DISTINCT(
    SELECT nodeto.document.title
    FROM link
    WHERE nodefrom.document.title =
        'Computing machinery and intelligence');

Using the schema in Figure 4.2 requires a more complex query expression.

DISTINCT(
    SELECT d2.title
    FROM document AS d1, document AS d2
    WHERE d1.title = 'Computing machinery and intelligence'
    AND (SELECT nodeid
        FROM UNNEST d2 ON contents)
    IN
    (SELECT nodeto
        FROM UNNEST ( UNNEST d1 ON contents)
    ON links));

Using the schema in Figure 4.4, requires an even more complex expression for the query.

SELECT title
FROM node AS n1
WHERE EXISTS (SELECT *
    FROM node
    WHERE title =
        'Computing machinery and intelligence'
    AND EXISTS (SELECT *
        FROM links
        WHERE nodeto = n.nodeid
    )
)
The above examples illustrate the problem of choosing one nesting at the logical level. The necessity, at the logical level, of having to choose between nestings should be avoided. The decision of which particular nesting to use should only be made at the physical level; we consider how to make this choice in Section 4.4.

4.2.3 Complex objects schemas in Atlas

The problem of nesting choice can be partially avoided, without duplicating entities, by storing the entity in one table and in other tables including references to that entity. In Atlas these reference attributes are specified by the keyword REF as described in Chapter 2. One approach to the problem of choosing nestings would be to use reference attributes in the logical schema to avoid duplication of data as in the schema in Figure 4.6. We could further simplify this schema by nesting authors inside document because the key for authors (and hence the reference to it) is the whole table and every author has written at least one document. Such a schema is shown in Figure 4.7.

Another approach to the problem of choosing nestings would be to use reference attributes in the logical schema and store the references in both directions as in the schema in Figure 4.8. We have introduced additional identifiers for the reference attributes, these avoid name clashes but require changes in some query expressions (substituting one identifier for another). Using this schema it is possible to commence a query on any entity in the schema and navigate along any of the relationships to any depth. Thus we could view the schema implied by the references as the infinite schema outlined in Figure 4.9.

Using this schema the simplest expression can be used for Query 4.1.

```
SELECT authors, contents[data]
FROM   document
WHERE  title = 'Computing machinery and intelligence';
```
CHAPTER 4 — SCHEMA DESIGN

```
document [  
docid INTEGER,  
title TEXT,  
authors [a REF author],  
contents [n REF node]  
] KEY = (docid)

author [  
firstname TEXT,  
surname TEXT  
] KEY = (firstname, surname)

node [  
nodeid INTEGER,  
data TEXT  
] KEY = (nodeid)

link [  
nodeto REF node,  
nodeto REF node,  
1kind TEXT,  
display TEXT  
]
```

Foreign key constraints

```
document.contents.links  
nodeto FOREIGN KEY REF document.contents
```

Template dependencies

```
document[  
docid ... contents[  
nodeid data link[ nodeto 1kind display]
]  
hypothesis U1 X1 V1 Y1 W1 Z1  
-----------------------------------------------  
conclusion U2 Y1 V2 X1 W2 Z2

...
```

```
document[ ... authors[a] ... ]  
author[firstname surname]  
hypothesis X1 Y1  
-----------------------------------------------  
conclusion (X1,Y1)
```

Figure 4.6: Database schema with simple acyclic references
document [
  docid    INTEGER,
  title    TEXT,
  authors  [firstname TEXT,
             surname TEXT],
  contents [n REF node]
] KEY = (docid)

node [
  nodeid   INTEGER,
  data     TEXT
] KEY = (nodeid)

link [
  nodefrom REF node,
  nodeto   REF node,
  lkind    TEXT,
  display  TEXT
]

\textbf{Template dependency}

\[
\begin{array}{cccc}
\text{hypothesis} & X_1 & Y_1 & W_1 & Z_1 \\
\hline \\
\text{conclusion} & Y_1 & X_1 & W_2 & Z_2 \\
\end{array}
\]

Figure 4.7: Minimal database schema with references
### Template Dependency

<table>
<thead>
<tr>
<th>link[ nodefrom nodeto lkind display ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>hypothesis</td>
</tr>
<tr>
<td>conclusion</td>
</tr>
</tbody>
</table>

Figure 4.8: Database schema with duplication of references
document [  
  docid INTEGER,  
  title TEXT,  
  authors [  
    firstname TEXT,  
    surname TEXT,  
    wrote [  
      docid INTEGER,  
      title TEXT,  
      authors [ ... ],  
      contents [ ... ] ] ],  
  contents [  
    nodeid INTEGER,  
    ofdoc (  
      docid INTEGER,  
      title TEXT,  
      authors [ ... ],  
      contents [ ... ]),  
    data TEXT,  
    links [  
      nodefrom (  
        nodeid INTEGER,  
        ofdoc ( ... ),  
        data TEXT,  
        links [...]),  
      nodeto (  
        nodeid INTEGER,  
        ofdoc ( ... ),  
        data TEXT,  
        links [...]),  
      lkind TEXT,  
      display TEXT ] ] ]

author [  
  firstname TEXT,  
  surname TEXT,  
  wrote [ ... ] ]

node [  
  nodeid INTEGER,  
  ofdoc ( ... ),  
  data TEXT,  
  links [... ] ]

link [  
  nodefrom ( ... ),  
  nodeto ( ... ),  
  lkind TEXT,  
  display TEXT ]

Figure 4.9: Virtual database schema implied by references
Similarly, for Query 4.2 we can use the simplest expression

\[
\text{DISTINCT}
\left( \begin{array}{l}
\text{SELECT nodeto.document.title} \\
\text{FROM link} \\
\text{WHERE nodefrom.document.title =} \\
\quad \text{'Computing machinery and intelligence'}
\end{array} \right)
\]

For some queries the expression on this schema will be simpler than the same query on any of the schemas in the Figures 4.1, 4.2, 4.3, 4.4, and 4.5. In general, all queries on this form of Atlas schema with bidirectional pointers can be expressed at least as simply as queries on the nested schemas.

### 4.2.4 General properties of complex object schemas

The results given in the previous sub-sections for a specific example of schema equivalence and simpler queries can be formalized. In our formalization we assume nested relations rather than nested sequences. There are three kinds of complex object schemas that we shall consider: *schemas without references* (as in Figure 4.2), *schemas with simple acyclic references* (as in Figure 4.6), and *schemas with bidirectional references* (as in Figure 4.8).

**schema without references**

Consider a complex object schema that contains only simple atomic attributes, nested tables and nested tuples. Such a complex object schema is a *schema without references*.

**schema with simple acyclic references**

Consider a complex object schema that contains only simple atomic attributes, reference attributes, and nested tables containing a single reference attribute. Such a complex object schema is a *schema with simple acyclic references* if the directed graph, that has vertices being tables and edges being reference attributes from one table to another table, is acyclic.
Consider a schema that contains only simple atomic attributes, reference attributes, and nested tables containing a single reference attribute. Such a schema is a **schema with bidirectional references** if for every reference from table $A$ to table $B$ there is an inverse reference from table $B$ to table $A$.

**Proposition 4.1** Any complex object schema without reference attributes can be represented by an equivalent schema with simple acyclic references.

This holds because there exists the following one-to-one mapping from the set of complex object schemas without references to the set of simple acyclic schemas.

1. Each nested table and nested tuple (in the schema without references) is mapped to a separate table (in the schema with simple acyclic references).

2. Each table (in the schema without references) containing a nested table is mapped to a table (in the schema with simple acyclic references) that contains a nested table consisting of a single reference attribute to the corresponding separate table.

3. Each table (in the schema without references) containing a nested tuple is mapped to a table (in the schema with simple acyclic references) that contains a reference attribute to a single tuple in the separate table containing the tuple.

**Proposition 4.2** Any query can be expressed as simply on either a complex object schema without references or an equivalent schema with simple acyclic references.

This holds because the simplest form of query on a complex object schema without references is a syntactically identical query on the equivalent simple acyclic schema, and vice versa.

**Proposition 4.3** Any query can be expressed at least as simply on a complex object schema with bidirectional references as on any equivalent complex object schema with simple acyclic references.
This holds because any query on a schema with bidirectional references can be expressed by a syntactically identical query on the equivalent simple acyclic schema. The converse of Proposition 4.3 is false.

The disadvantage of the schema with bidirectional references is that references are duplicated, which can create update anomalies at the logical level and inefficiencies at the physical level. In the next section we use the logical data model for complex objects that was introduced in Chapter 2. Logical schemas in this data model have the flexibility of the references and can be mapped into any of several physical schemas without necessarily duplicating data or references and without creating update anomalies or inefficiencies.

### 4.3 Logical schemas based on complex objects

In designing schemas for structurally objected-oriented database systems it is easy to forget the distinction between the logical and physical schemas. This creates problems for schema optimization as the best logical schema is not necessarily the best physical schema. Logical schemas should retain as many links as is possible, to make queries as simple as possible. Physical schemas should only retain links which allow queries to be answered as efficiently as possible.

We require the logical schema to allow the flexibility of the schema in Figure 4.9 but not require the reference links to be fully specified as in Figure 4.8; which references should be stored is an issue for the physical level. In Section 2.5.2 we defined a notation for such a logical schema and in this section we examine how to generate such a schema given a semantic data model. The schema in Figure 2.11 is equivalent to the schema in Figure 4.1 if some additional constraints are added as in the schema in Figure 4.10.

On the schema in Figure 4.10 we once again can use the simplest expressions of the queries; for example, the following simple expression for Query 4.1.
```sql
document [
    docid INTEGER,
    title TEXT,
    authors LIST OF author,
    contents LIST OF node
] KEY = (docid)

author [
    firstname TEXT,
    surname TEXT,
    wrote SET OF document
] KEY = (firstname, surname)

node [
    nodeid INTEGER,
    ofdoc document,
    nkind TEXT,
    data TEXT,
    links SET OF link
] KEY = (nodeid)

link [
    nodefrom node,
    nodeto node,
    lkind TEXT,
    display TEXT
]

document.authors INVERSE OF author.wrote
document.contents INVERSE OF node.ofdoc
node.links INVERSE OF link.nodefrom
node.links INVERSE OF link.nodeto
```

Cardinality constraint

NOT EMPTY (author[wrote])

Figure 4.10: Logical database schema based on complex objects
SELECT node[data], authors
FROM document
WHERE title = 'Computing machinery and intelligence';

Similarly for Query 4.2 we are able to use the following simple expression.

DISTINCT(
    SELECT nodeto.document.title
    FROM link
    WHERE nodefrom.document.title =
    'Computing machinery and intelligence'
)

The simplest expression can always be used on a logical schemas.

**Proposition 4.4** Any query can be expressed as simply on either a logical schema based on complex objects or on an equivalent complex object schema with bidirectional references.

This holds because a query on a logical schema based on complex objects can be expressed by a syntactically identical query on the equivalent complex object schema with bidirectional references, and the simplest form of a query on complex object schema with bidirectional references can be expressed by a syntactically identical query on the equivalent logical schema based on complex objects.

If we define the logical schema using an appropriate view on the physical schema these queries can easily be translated into the appropriate query at the physical level. This issue of how such a logical schema may best be mapped to a physical schema we consider further in Section 4.4.

The E-R model is one of several possible semantic data models which could be used as a starting point for the design of logical schemas.

The following schema translation algorithm translates a schema in the E-R model to our logical model based on complex objects. We assume that in the E-R diagram no attributes are attached to relationship types; this may involve an initial simple
transformation of the E-R diagram to eliminate any such attributes which could involve introducing additional entity types.

1. Map each entity type in the E-R model to an object class in the logical model.

2. For each multivalued attribute in an entity type in the E-R model add an object class in the logical model, and add a multiple-object-reference attribute to the object class in the logical model corresponding to the entity type.

3. For each atomic attribute in the E-R model add an attribute to the corresponding object class in the logical model.

4. For each 1:1 relationship in the E-R model add an object-reference attribute to both of the participating object classes in the logical model.

5. For each 1:N relationship in the E-R model add an object-reference attribute to the object class in the logical model on the “N” side, and add a multiple-object-reference attribute to the object class (LIST OF or SET OF) in the logical model on the “1” side.

6. For each N:M relationship in the E-R model add a multiple-object-reference attribute to both of the participating object classes in the logical model.

7. For each pair of attributes added to the logical model in steps 3 to 6 add an INVERSE OF constraint.

8. For each n-ary relationship in the E-R model add a multiple-object-reference attribute to the participating object classes in the logical model on any “1” side and an object-reference attribute to the object class in the logical model on the “N” side. Add appropriate INVERSE OF constraint.

As an example of the schema translation algorithm consider the E-R diagram in Figure 2.2. Before applying the algorithm we must transform the E-R diagram to attach the attributes lkind and display to an entity type; this can be achieved by adding a new entity type link. The resulting E-R diagram is shown in Figure 4.11.
Figure 4.11: Ammended E-R diagram
The first step maps the entity types document, node, and link to object classes. The second step adds the object class author, and adds the multiple-object-reference attribute authors to the object class document. The third step adds atomic attributes to each object class: docid and title to document, nodeid and data to node, lkind and display to link, and firstname and surname to author. As there are no 1:1 relationships the fourth step does not apply. The fifth step adds the following pair of attributes: contents in document and ofdoc in node. As there are no N:M relationships the sixth step does not apply. The seventh step adds three INVERSE OF constraints. The eighth step adds the following pairs of attributes: links in node and nodefrom in link, links in node and nodeto in link together with the last INVERSE OF constraint.

4.4 Physical schemas based on complex objects

Choosing the underlying physical representation is a separate problem to choosing a schema that allows ease of query expression at the logical level. At the physical level the nested logical schemas can be mapped to a variety of nested relational schemas. Scholl [129] has explored some of the options for physical schemas using tuple identifiers (physical address of an object) and object identifiers (stored values which represent an object, these values are surrogates generated by the system). Our approach is simpler, but less object-oriented, in that it relies on key values. Any of the schemas in Figures 4.1, 4.2, 4.4, 4.7, or 4.8 could be used as a physical schema for representing Schema 2.11, and these are not the only choices available.

The problem of physical representation has been considered in the object-oriented paradigm. For example, Rumbaugh et al. [114] describe three ways to physically implement an association between objects. Two of these correspond to the schemas in Figure 4.6 and Figure 4.8. The third approach represents the association in a separate table; such a schema is illustrated in Figure 4.12.

In general nested relations have an advantage over flat relations at the physical level. Flat relations are inappropriate at the physical level if queries frequently require decomposed entities to be rejoined. In our hyperbase example, if the physical
Figure 4.12: Database schema with separate table for associations

document [  
    docid INTEGER,  
    title TEXT,  
  ] KEY = (docid)

author [  
    authid INTEGER,  
    firstname TEXT,  
    surname TEXT,  
  ] KEY = (authid)

node [  
    nodeid INTEGER,  
    data TEXT,  
  ] KEY = (nodeid)

link [  
    linkid INTEGER,  
    lkind TEXT,  
    display TEXT  
  ]

doc_auth [  
    docid INTEGER,  
    authid INTEGER  
  ]

doc_node [  
    docid INTEGER,  
    nodeid INTEGER  
  ]

node_link [  
    nodefrom INTEGER,  
    nodeto INTEGER,  
    linkid INTEGER  
  ]
design reflects a logical schema that consists of normalized relations, then the data will be stored in four physical tables. This will make many common queries expensive to evaluate; for example, to retrieve all the information about a single document, including the information to display links, four relations rather than a single table must be accessed.

One approach for overcoming this problem is to pre-join the offending entities and no longer have the database fully normalized at the physical level [46]. Another approach is to store the data as complex objects based on nested relations [109, 118, 157]. In some systems data clustering overcomes this problem; for example, Scholl et al. [130] represent data clustering in a nested relational kernel.

How do we choose the best physical schema and, given that choice, how do we map queries and updates from our logical schema? Before addressing the optimization issue of selecting the best physical schema we first consider how to map queries and updates from our logical schema into a given physical schema.

### 4.4.1 Mapping queries and updates

Mapping queries from the logical schema into the physical schema is straightforward if we define the logical schema in terms of a view on the physical schema. To allow queries that can follow references to any depth, we need to introduce derived attributes into the tables in the physical schema.

We propose derived attributes as a possible extension to TQL similar to the attributes of type “postquel” query in Postgres [113, 134]. Derived attributes are parameterized procedure-types; within a derived attribute $ refers to the tuple of the relation on which a value for the derived attribute is being computed. Consider the following example schema.

```perl
letter [
    authors [firstname TEXT, surname TEXT],
    surnames := SELECT surname FROM $.authors,
    :
]
```
For each tuple \( z \) in the relation \texttt{letter} a value for the derived attribute \texttt{surnames} is generated by applying the query

\[
\text{SELECT surname FROM $.authors}
\]

where \( \$ \) is replaced by the tuple \( z \) in \texttt{letter}.

If we had chosen the schema in Figure 4.7 as the physical representation we would need to add a view defining the object class \texttt{author} and a number of derived attributes to each table in the physical schema as shown in Figure 4.13. Similar transformations can be made for the other physical schemas.

Updates on attributes actually stored in the physical schema are straightforward, for example to add an author to the document identified as ‘1’.

\[
\text{UPDATE document}
\text{SET (INSERT authors VALUES ['John','Smith'])}
\text{WHERE docid = 1}
\]

Updating other attributes is the problem of view updates, which has been dealt with extensively in the literature (for instance [9, 10, 18, 36, 43]). This is an area for further investigation, as the inverse relationships may provide a mechanism for determining the update rules.

### 4.4.2 Optimal physical schema

Selecting the “best” physical schema for a given logical schema is an optimization problem based on a particular pattern of queries.

The optimization procedure relies on a method of representing generic query forms on the particular database. For example, a generic query, of which Query 4.1 is a specific case, is given as follows:

\[
\text{SELECT node[data], authors}
\text{FROM document}
\text{WHERE title CONTAINS $1}
\]
Figure 4.13: Logical database schema based on physical schema
For each generic query form we maintain a weighting. The weighting could be pre-assigned on expected database usage or based on actual usage or based on importance of quick response. If the weightings are assigned on actual usage the weighting could be a simple count of the number of times each query form is used. A low weight corresponds to an infrequent query, a high weight corresponds to a frequent query.

The good physical schema design algorithm will optimize database performance. We assume there is a procedure for estimating the cost of a query, including the update queries. Using the costs of individual queries we are able to minimize the overall query cost as follows.

$$\text{overall-cost} = \sum_{q_i \in Q} \text{query-cost}_i \times \text{query-weighting}_i$$

where each query form $q_i$ is an element of a set of queries forms $Q$, each $q_i$ has a retrieval cost of query-cost$_i$, and each $q_i$ has a weighting query-weighting$_i$.

If we allow duplication of data then we need an additional constraint, the size of the database, as there are potentially an unlimited number of duplications possible which will aid the performance of some queries.

The problem of an optimal nested structure has been investigated — Haffez and Ozsoyoglu [60] examined join costs whereas Sacks-Davis et al. [119] and Wen et al. [150] have studied both indexing and join costs. Both approaches assume that a hierarchical structure is already defined. Our logical schemas provide a method for defining that initial hierarchical structure on which their optimization methods could be applied.

### 4.5 Summary

There are several issues affecting the design of databases with complex objects. Logical schema design of complex objects should allow queries to be expressed in as simple a manner as possible — with each entity being modelled as an independent object, whereas physical schema design of complex objects should allow queries to be executed as efficiently as possible. The selection of the ideal nesting of entities at
the physical level is an optimization problem — finding the structure which allows
the most efficient querying.

We have described an approach for the logical design of complex objects based
on the entities on which queries may be made. Aspects of the model we have used
are similar to the object-oriented data model described by Schek and Scholl [126]
which is an evolution of the relational and network data models. Logical schema
design of complex objects should allow queries to be expressed in as simple manner
as possible. This is a different problem from the physical implementation of complex
objects. Nested relations provide an appropriate physical implementation for the
physical schema. The choice of the best physical schema is an optimization problem
which takes into account the efficiency of query processing.

The central theme is that it is desirable that the logical design of complex objects
be based on entities on which queries be made and that the complex objects at the
physical level may differ from the complex objects at the logical level.

The issues discussed in this chapter and the model proposed are more general
than just document database design and can be applied to any database system
based on complex objects, including object-oriented database systems.
Chapter 5

Text Properties

An important component of the optimization problem defined in the previous chapter is calculating the cost of queries for each schema. Without building each physical schema for various databases and comparing the actual costs we can estimate these costs using mathematical formulas. In this chapter we do not derive these formulas (that is left until Chapter 6), instead we develop a mathematical model of text which will be used in these formulas.

It is common to model the distribution of words in text by measures such as the Poisson approximation. However, these measures ignore effects such as clustering: our analysis of document collections demonstrates that the Poisson approximation can significantly overestimate the probability that a document contains a word. Based on our analysis, we propose a new model for distribution of words in text, and show how this model can be used to estimate the probability that a document contains a word and the number of distinct words in a document.

Models for the distribution of words in text are used to estimate the performance of algorithms for text compression and for full text retrieval from databases. Many of these models, such as the Zipf and lognormal distributions, are used to predict the distribution of word frequencies [24, 151, 154, 155]. However, such measures do not predict the probability that a given document in a document collection contains a particular word. Other models, such as the Poisson distribution, can be used to estimate this probability, but such models are inaccurate, as they ignore effects such
as *word clustering*.

Clustering arises from the fact that words tend to be repeated a number of times in the same piece of text, even words that are (overall) quite rare. For example, the effectiveness of adaptive text compression techniques depends in part on clustering effects, where in general the number of bits used to encode a word decreases as the number of recent occurrences of the word increases [30, 88]. The greater the clustering the more compression can be achieved.

Clustering has not been quantified before. Without some allowance for clustering, formulas for estimating the average performance of different physical schemas for a document database will be less accurate. For example, in full text retrieval systems, the cost of answering a query depends on the number of documents that satisfy the query, and the size of an index depends on the number of unique terms in each document.

To obtain accurate estimates of the probability that a document of a given size contains a given word, we analysed several document collections. We describe the methods and results of our analysis of document collections in Section 5.1. In Section 5.2 we discuss existing models for distribution of words in text; our analysis reveals that the Poisson estimate of the probability that a document of a given size contains a given word can be much greater than the observed probability. To derive a more accurate model we use the regression techniques described in Section 5.3. In Section 5.4 we propose, as a new measure, a *clustering model* that generally provides a good fit to observed data. Like other accurate models of text [151], this model is empirical.

We discuss how the clustering model can be verified and applied in Section 5.5; we show that the clustering model can be used to estimate the average number of documents containing a word of a given probability and to estimate the number of distinct words in a document. In Chapter 6, we estimate index sizes for different ways of storing text in databases; index sizes are dependent on the average number of distinct terms in each document. We also use the clustering model to approximate the probability that a set of terms appears in a document. This probability is useful because it can be applied to estimation of the number of answers to a query.
In addition to the analysis of alternative physical schemas, the clustering model has been used to estimate the importance of query terms in information retrieval by Wallis et al. [147]. It has potential applications in the analysis of ranking strategies, indexing and storage schemes, data compression techniques, and even literary criticism in comparing writing styles. In Section 5.6 we discuss the limitations of the clustering model. The final section summarizes the distinctive properties of the clustering model and types of problems it can be used to solve.

There is glossary on page 210 of the notation used in this chapter and the next chapter.

5.1 Analysis of document collections

In this section we discuss how we analysed our document collections and present the results of our analysis. For the purposes of our analysis, we assumed that a word is a series of alphabetic characters flanked by non-alphabetic characters. As in many information retrieval systems, uppercase and lowercase letters were converted to a single case, and the resulting strings were stemmed using Lovins’s algorithm [82], thus reducing the number of distinct terms under consideration. We also assumed that a document is an entire, contiguous piece of text such as a book of the Bible, an Act of Parliament, or a speech, and that a document collection is a set of documents from a single source. The size of a document was measured by the number of words in the document rather than by the number of characters.

We had several document collections available for analysis. We chose to concentrate on three, the King James version of the Bible, the Commonwealth Acts of Australia from 1901 to 1988 (or Comact), and an extract from the 1989 Western Australian Hansard, the official transcript of that state’s parliamentary proceedings. The sizes of these collections are shown in Table 5.1. As can be seen, the Bible is much smaller than both Comact and Hansard, and was principally used to verify results obtained from the latter two collections. The presentation of our results in this chapter is principally based on Comact. The three collections have different characteristics. The Bible is somewhat representative of written English. Hansard
is a collection of transcripts of spoken English, and each document (which is a debate on a single topic) tends to be repetitive and informal in structure. In contrast, the language in Comact is highly formalized, the vocabulary limited, and the text available to us contained a large number of spelling errors, amounting to approximately one-third of all distinct terms. However, within each collection documents are written in a fairly homogeneous style.

Our principal aim was to discover a relationship between: the probability that a given word occurs; document size; and the probability that a document of that size contains that word. For most sizes, however, our document collections contained very few documents of or near that size. Thus for our analysis we chose to select a range of possible lengths and construct pseudo-documents (or fragments) of each length. Each pseudo-document was an excerpt of contiguous text from a real document. We generated the pseudo-documents by taking several fragments of each length from each real document. If the originating document was shorter than a given length, no fragment of that length was taken from it. This method of constructing pseudo-documents of given lengths from real documents allowed us to simulate collections of independent documents of a given size drawn from a large document space. We show in Section 5.5 that the model derived from pseudo-documents can be used to predict aspects of the behaviour of real documents.

The number of fragments of each length was chosen so that about 5% of each document was represented in fragments of that length. The starting point of each fragment was chosen at random, and we did not exclude the possibility of overlap. We considered 26 different fragment lengths between 2 and 100,000 words. For

Table 5.1: Size of each document collection

<table>
<thead>
<tr>
<th></th>
<th>Number of documents</th>
<th>Number of distinct words</th>
<th>Number of word occurrences</th>
<th>Average document length (in words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bible</td>
<td>66</td>
<td>8,892</td>
<td>791,448</td>
<td>11,992</td>
</tr>
<tr>
<td>Comact</td>
<td>3,459</td>
<td>21,740</td>
<td>16,548,025</td>
<td>4,784</td>
</tr>
<tr>
<td>Hansard</td>
<td>13,545</td>
<td>22,484</td>
<td>4,473,791</td>
<td>330</td>
</tr>
</tbody>
</table>

---

References:

[1] Author(s), Title of the Work, Publisher, Year of Publication.

[2] Author(s), Title of the Work, Publisher, Year of Publication.

[3] Author(s), Title of the Work, Publisher, Year of Publication.
some of these lengths, the number of fragments generated for each of the document collections is shown in Table 5.2.

<table>
<thead>
<tr>
<th></th>
<th>Fragment length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
</tr>
<tr>
<td>Bible</td>
<td>1 021</td>
</tr>
<tr>
<td>Comact</td>
<td>22 530</td>
</tr>
<tr>
<td>Hansard</td>
<td>9 741</td>
</tr>
</tbody>
</table>

Table 5.2: Number of fragments in each document collection

Word occurrence probabilities for each document collection were computed by counting word occurrences over the whole of the document collection. The probabilities of fragments containing a word were given by counting, for each fragment size, the number of fragments of that size in which the word occurred. These probabilities were computed by maintaining a splay tree [133] of distinct words with, for each word, several counters.

Throughout this chapter, we use $n$ to denote the number of words in a fragment, $p(w)$ or just $p$ to denote the probability that a randomly chosen word in the document collection is the word $w$ (that is, the occurrence probability of $w$), and $p_n(w)$ or just $p_n$ to denote the probability that a fragment of length $n$ contains word $w$. For all words in Comact, $p$ and $p_n$ (where $n = 100$, 1 000, and 10 000) are related as shown in Figure 5.1. We have omitted from Figure 5.1 the few points where $p > 0.005$.

One problem with our method of estimating probabilities is the difficulty of accurately deriving extreme $p$ and $p_n$ values. At very low $p$ values only a few distinct $p_n$ values occur, representing words occurring in only a few fragments; thus, at such $p$ values, patterns such as simple lines appear in graphs. Moreover, since many of the words with very low $p$ did not occur in any of the fragments we generated, our method derived $p_n = 0$, which is clearly an error since all words with some probability of occurrence must have non-zero $p_n$ for all finite $n$. A similar but less severe problem occurs for very high $p$ values, as words with such $p$ values occur in
Figure 5.1: Relationship between $p$ and $p_n$ on Comact
nearly every fragment. Thus, estimated probabilities with very low or very high \( p \) or \( p_n \) were substantially inaccurate. In deriving our model in Section 5.4, we eliminate these extreme values for \( p \) and \( p_n \).

5.2 Existing models of word distributions

5.2.1 Models of word probabilities

Zipf’s law is perhaps the best known model of word probabilities. It describes the fact that when words are ranked on frequency, from most to least frequent, plotting rank against frequency yields a hyperbolic curve [154, 155]. However, it has been argued that too much emphasis has been placed on this result: even words produced by a simple random generator conform to Zipf’s law [151].

In any case, Zipf’s law, or amendments to Zipf’s law such as that proposed by Mandelbrot [86], do not apply to the problem we are considering: the number of word occurrences and number of distinct words in a document collection do not specify the parameters of the Zipf curve. Nor do these parameters, if known, help determine the probability that a document contains a given term. Although theoretically elegant, Zipf’s law provides only a loose fit to actual text, and in practice must be modified by introduction of additional parameters [151]. The lognormal distribution [24] has similar limitations [151].

5.2.2 The Poisson approximation

For a sequence of trials in which the probability of each outcome is unchanged between trials (that is, the trials are equivalent and independent), the probability that exactly \( v \) of the trials have a particular outcome is given by the binomial distribution. Where the number of trials is large, the Poisson approximation can be used to estimate this probability using the formula for the Poisson random variable \( X \):

\[
P(X = v) = e^{-\lambda} \frac{\lambda^v}{v!}
\]
where \( \lambda \) is the mean number of trials with the desired outcome \([50]\). Thus the probability that at least one trial has a particular outcome is estimated by

\[
P(X \geq 1) = 1 - e^{-\lambda}
\]

In our context, the probability that a fragment contains one or more occurrences of a particular word can be estimated using the Poisson approximation by

\[
pois_n = 1 - e^{-n \cdot p}
\]

where \( pois_n \) is the Poisson-estimated approximation to \( p_n \). For fragments of sizes \( n = 100, n = 1000, \) and \( n = 10000 \), \( pois_n \) is shown as the top curve (dashed line) in Figures 5.2, 5.3, and 5.4 respectively. Observed \( p_n \) values for each word are shown as points in these figures. These graphs show that the Poisson estimate \( pois_n \) significantly overestimates \( p_n \) for most values of \( p \). As can be seen, the error in the Poisson estimate becomes greater as \( n \) increases.

Although the Poisson approximation can be used to model distribution of words in text, choice of words when speaking or writing is not a sequence of independent trials. Usually, choice of a word is strongly limited by the words preceding it \([151]\): for example, ‘choice’ is quite likely to be followed by ‘of’, but it is most unlikely that ‘choice’ would be followed by ‘the’. For non-text data, it can be assumed that data values are randomly distributed \([153]\), so that numbers of matching records can be estimated via the binomial distribution and hence the Poisson approximation. Our results indicate that this assumption is invalid for records containing text. Christodoulakis has shown that, where this assumption is invalid, estimates based on the assumption over-estimate the number of matching records \([28]\). Kent et al. \([67]\) and Sacks-Davis et al. \([116]\) implicitly assumed that words in text data are randomly distributed; although for their work this over-estimate led to a more conservative design for their indexes and an improvement in worst-case performance.

Poisson estimates of the probability that a document contains a word are usually an overestimate, as they are based on the assumption that words are evenly distributed in text. Under this assumption, a rare word that has occurred in a document is very unlikely to occur elsewhere in that document. In practice, however,
Figure 5.2: Observed, Poisson-estimated, and cluster-estimated $p_n$ for $n = 100$ on Comact
Figure 5.3: Observed, Poisson-estimated, and cluster-estimated $p_n$ for $n = 1000$ on Comact
Figure 5.4: Observed, Poisson-estimated, and cluster-estimated $p_n$ for $n = 10000$ on Comact
if any word occurs in a document, the probability that it will occur again, possibly several times, is relatively high. Thus, since a word is likely to occur several times in each document in which it occurs at all, Poisson estimates based on occurrence counts across a document collection will in general be too high.

### 5.3 Regression

Empirically determining a model of word clustering requires that we find equations which fit the data in Figures 5.2, 5.3, and 5.4. We do this in Section 5.4 using the regression techniques described in this section.

We first describe linear regression (or Gaussian least squares curve fitting) using the notation of Devor [45]. Suppose we have: an independent variable \( x \) and a dependent variable \( y \); a series of \( x \) values \( x_1, \ldots, x_m \); and a corresponding series of \( y \) values \( y_1, \ldots, y_m \). For a probabilistic model in which there exists parameters \( \beta_0 \) and \( \beta_1 \), for any fixed value of the independent variable \( x \)

\[
y = \beta_0 + \beta_1 \cdot x + \epsilon
\]

where \( \epsilon \) is a random variable with mean zero and variance \( \sigma^2 \). The equation

\[
y = \beta_0 + \beta_1 \cdot x
\]

is called the *true regression line*. We assume the pairs \((x_i, y_i)\) are distributed about the true regression line in a random manner.

The principle of least squares states that among all straight lines \( y = \beta_0 + \beta_1 \cdot x \) the least squares line or estimated regression line

\[
y = \hat{\beta}_0 + \hat{\beta}_1 \cdot x
\]

is that line which minimizes the sum of the squared deviations

\[
\sum (y_i - (\hat{\beta}_0 + \hat{\beta}_1 \cdot x_i))^2
\]

where \( \hat{\beta}_0 \) and \( \hat{\beta}_1 \) are point estimates for \( \beta_0 \) and \( \beta_1 \). The co-efficients \( \hat{\beta}_0 \) and \( \hat{\beta}_1 \) which minimize the sum of the squared deviations are

\[
\hat{\beta}_1 = \frac{m \cdot \sum x_i \cdot y_i - (\sum x_i) \cdot (\sum y_i)}{m \cdot \sum x_i^2 - (\sum x_i)^2}
\]
\[ \hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \cdot \sum x_i}{m} \]

To assign a level of confidence to the coefficients, an estimate of the variance \( \hat{\sigma}^2 \) must be found, namely

\[ \hat{\sigma}^2 = \frac{\sum (y_i - (\hat{\beta}_0 + \hat{\beta}_1 \cdot x_i))^2}{m - 2} \]

A confidence interval of \( 100 \cdot (1 - \omega) \%) \) for \( \beta_0 \) is given by

\[ \hat{\beta}_0 \pm t_{\omega/2, m-2} \cdot \hat{\sigma} \cdot \sqrt{\left( \frac{1}{m} + \left( \frac{\sum x_i}{m} \right)^2 \cdot \frac{1}{\sum x_i^2 - (\sum x_i)^2/m} \right)} \]

A confidence interval of \( 100 \cdot (1 - \omega) \%) \) for \( \beta_1 \) is given by

\[ \hat{\beta}_1 \pm t_{\omega/2, m-2} \cdot \hat{\sigma} \cdot \frac{1}{\sqrt{\sum x_i^2 - (\sum x_i)^2/m}} \]

In these equations, \( t_{\omega/2, m-2} \) is the Student’s \( t \) distribution\(^1\) with \( m - 2 \) degrees of freedom; for a confidence interval of 95\%, \( t_{\omega/2, m-2} \) is approximately 1.960 for large \( m \).

Where there is a non-linear relationship between the independent and dependent variables we need to use non-linear regression. If we represent the non-linear variables by \( x' \) and \( y' \) and there exists functions \( f \) and \( g \) so that there is a linear relationship between \( f(x') \) and \( g(y') \) then the above equations for linear regression can be transformed for non-linear regression by setting \( x = f(x') \) and \( y = g(y') \). The parameters \( \hat{\beta}_0 \) and \( \hat{\beta}_1 \) can be estimated by substituting transformed values \( x_i \)'s and \( y_i \)'s into the above formulas, and approximate confidence intervals for \( \hat{\beta}_0 \) and \( \hat{\beta}_1 \) can be derived in a similar fashion.

### 5.4 The clustering model

If a word occurs in a document there is a relatively high probability that it will occur more than once in that document. We call this effect word clustering. In this section

\(^1\)“A random variable has Student’s \( t \) distribution with \( n \) degrees of freedom if it has the same distribution as the quotient \( u \cdot \frac{\sqrt{v}}{v} \), where \( u \) and \( v \) are independent random variables, \( u \) having a normal distribution with mean 0 and standard deviation 1, \( v^2 \) having a \( \chi^2 \) distribution with \( n \) degrees of freedom.” [23]
we describe an empirically-determined *clustering model* that relates the probability that a word occurs, document size, and the probability that a document of that size contains the word. This model is not based on a psychological or linguistic theory of language: rather, it is an approximation based on observations of the properties of actual text.

For most words $w$, and for all but small document lengths $n$, the observed probability $p_n(w)$ is substantially less than $\text{pois}_n(w)$. The degree of difference between $p_n(w)$ and $\text{pois}_n(w)$ depends on $w$, since words with the same probability of occurrence can occur in different numbers of documents. The degree to which the distribution of a word differs from a random distribution has been described by Wallis et al. [147] as the *topic specificity* of the word.

As can be seen from Figures 5.2, 5.3, and 5.4, the distribution of $(p, p_n)$ values is similar in shape to that predicted by the Poisson approximation. We therefore assumed that a good fit would be given by models of similar form to Poisson. Thus we investigated models of the form

$$clus_n = 1 - e^{-\psi(n,p)}$$

where $\psi$ is a function of $n$ and $p$, and $clus_n$ is the cluster-estimated approximation to $p_n$. In order for $clus_n$ to be less than $\text{pois}_n$ we require $\psi(n, p)$ to be less than $n \cdot p$. Based on our observations of text, this relationship should hold for all but small documents.

We investigated several $\psi$ functions. The simplest form of $\psi$ that gave a good fit to our data is

$$\psi(n, p) = \alpha^{1-\beta} \cdot n^\beta \cdot p$$

where $\alpha$ and $\beta$ are constants for a particular document collection.

In $\psi$, the parameter $\alpha$ is the document size above which clustering comes into effect. When $n = \alpha$, $clus_n$ and $\text{pois}_n$ are equal. Our formula predicts less clustering than the Poisson approximation when $n < \alpha$. Small documents or fragments, such as this slightly contrived sentence, usually exhibit little clustering and may actually show an opposite effect, because the authors’ tend to avoid repeating words in any short piece of text.
The parameter $\beta$ measures the degree of clustering; the smaller $\beta$ is, the more tightly words are clustered. When $\beta = 1$, there is no clustering and our formula reduces to the Poisson approximation. In general $\beta < 1$, so if $n > \alpha$ then $(\alpha/n)^{1-\beta} < 1$ and $\psi(n, p) < n \cdot p$. The smaller $\beta$ is, the greater the difference between $\text{clus}_n$ and $\text{pois}_n$. Therefore, for collections that are accurately modelled by $\text{clus}_n$ with small $\beta$, clustering is high: in such collections, words tend to occur frequently in a small number of documents, and are not evenly distributed throughout the collection.

The parameters $\alpha$ and $\beta$ will vary between collections because they will have different points at which clustering begins to take effect, and because different styles of text will cluster to different degrees. For a particular document collection, $\alpha$ and $\beta$ can be estimated as follows. Let $k = \alpha^{1-\beta} \cdot n^\beta$, which is a constant for given $n$. Inverting the formula for $\text{clus}_n$, and using observed values for $p_n$, we get

$$k = -\frac{\log_e(1 - p_n)}{p}$$

for $p > 0$ and $p_n < 1$. In Figure 5.5 we have graphed $k$ against $p_n$ for Comact fragments of size 1 000. For all but extreme values of $p_n$ (which, as discussed in Section 5.1, are difficult to derive using our method), $k$ values generally fall between 200 and 700, and are independent of $p_n$ values.

In Figure 5.6, median $k$ values for each fragment size $n$ are plotted on a logarithmic scale, as suggested by Daniel and Wood’s text [38]. As can be seen, the relationship between $\log_e(n)$ and $\log_e(\text{median } k)$ is near linear, justifying our assumption that $k = \alpha^{1-\beta} \cdot n^\beta$ for some $\alpha$ and $\beta$. Using non-linear regression as described in Section 5.3, the solid, straight line can be fitted to the $(\log_e(n), \log_e(k))$ points. The equation of the line is

$$\log_e(k) = \hat{\beta}_0 + \hat{\beta}_1 \cdot \log_e(n)$$

Its intercept with the $\log_e(k)$ axis is $\hat{\beta}_0$, and its slope is $\hat{\beta}_1$. Taking exponents we get

$$k = e^{\hat{\beta}_0} \cdot n^{\hat{\beta}_1}$$

Hence $\beta = \hat{\beta}_1$ and

$$\alpha = e^{\hat{\beta}_0/(1-\hat{\beta}_1)}$$
Figure 5.5: $k = -\frac{\log(1-p_n)}{p}$ against $p_n$ for $n = 1000$ on Comact
Figure 5.6: Median $k = -\frac{\log_e(1-p_n)}{p}$ against $n$ on Comact
which is the intercept of the clustering and Poisson lines.

Note that in Figure 5.6 the median is based on \( p_n \) values in the range \( 0.25 \leq p_n \leq 0.75 \), since as discussed above estimates for \( p_n \) values outside this range are inaccurate. Also, this graph contains points for values of \( n \) not shown elsewhere in this chapter, and points for \( n < 60 \) or \( n > 40000 \) have been discarded since at these points either there were less than twenty fragments, or there were less than twenty words with \( p_n \) values in the range 0.25 to 0.75.

On Comact, the above method for deriving \( \alpha \) and \( \beta \) yields \( \alpha = 40.8 \) and \( \beta = 0.734 \). For these figures, the estimate \( \text{clus}_n \) is graphed (continuous line) for Comact fragments of sizes \( n = 100, n = 1000, \) and \( n = 10000, \) in Figures 5.2, 5.3, and 5.4 respectively. A summary of \( \alpha, \beta, \) and \( \alpha^{1-\beta} \) values and confidence intervals for each document collection is given in Table 5.3. To derive this table, the techniques

<table>
<thead>
<tr>
<th></th>
<th>( \alpha )</th>
<th>( \beta = \hat{\beta}_1 )</th>
<th>( \alpha^{1-\beta} = e^{\hat{\beta}_0} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>least squares estimate</td>
<td>95% confidence interval</td>
<td>least squares estimate</td>
</tr>
<tr>
<td>Bible</td>
<td>2.74</td>
<td>[0.90,8.33]</td>
<td>0.920</td>
</tr>
<tr>
<td>Comact</td>
<td>40.8</td>
<td>[25.2,66.0]</td>
<td>0.734</td>
</tr>
<tr>
<td>Hansard</td>
<td>6.28</td>
<td>[1.89,20.8]</td>
<td>0.914</td>
</tr>
</tbody>
</table>

Table 5.3: \( \alpha \) and \( \beta \) values for each document collection

given in Section 5.3 have been used to give 95% confidence intervals for \( \hat{\beta}_0 \) and \( \hat{\beta}_1 \). The ranges of values for \( \alpha \) derive from the confidence in \( \hat{\beta}_0 \) and \( \hat{\beta}_1 \). Although the \( \alpha \) values have substantial variation, the value \( \alpha^{1-\beta} \) used in \( \psi \) is much more tightly contained, indicating that significant errors in estimation of \( \alpha \) only marginally affect the accuracy of the clustering model. In Figure 5.6 we have graphed the upper and lower 95% bounds, and as can be seen these lines enclose a small region.

There are marked differences between the \( \alpha \) and \( \beta \) values for the different document collections; these probably correspond to variations in literary style in the
different collections. We have not analysed a large enough number of collections to prescribe typical $\alpha$ and $\beta$ values, but analysis of further collections has indicated that values near of those of Hansard appear to be common.

A comparison of mean observed $p_n$ values for Comact with the estimates $pois_n$ and $clus_n$ is shown in Table 5.4. As can be seen, cluster-model estimates of $p_n$ are always closer to the observed values than are the Poisson estimates.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$p$</th>
<th>Mean $p_n$</th>
<th>$pois_n$</th>
<th>$clus_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>$10^{-6}$</td>
<td>0.000075</td>
<td>0.000100</td>
<td>0.000079</td>
</tr>
<tr>
<td></td>
<td>$10^{-4}$</td>
<td>0.0062</td>
<td>0.0100</td>
<td>0.0078</td>
</tr>
<tr>
<td></td>
<td>$10^{-2}$</td>
<td>0.55</td>
<td>0.63</td>
<td>0.54</td>
</tr>
<tr>
<td>1000</td>
<td>$10^{-6}$</td>
<td>0.00056</td>
<td>0.00100</td>
<td>0.00043</td>
</tr>
<tr>
<td></td>
<td>$10^{-4}$</td>
<td>0.033</td>
<td>0.095</td>
<td>0.042</td>
</tr>
<tr>
<td></td>
<td>$10^{-2}$</td>
<td>0.95</td>
<td>1.0</td>
<td>0.99</td>
</tr>
<tr>
<td>10000</td>
<td>$10^{-6}$</td>
<td>0.0044</td>
<td>0.0100</td>
<td>0.0023</td>
</tr>
<tr>
<td></td>
<td>$10^{-4}$</td>
<td>0.20</td>
<td>0.63</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>$10^{-2}$</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 5.4: Typical $p$, $p_n$, $pois_n$, and $clus_n$ values on Comact

### 5.5 Verifying the clustering model

As can be seen in Figures 5.2, 5.3, and 5.4, the clustering model provides a reasonably close fit to observed data. Another way of examining the accuracy of the clustering model is to consider its predictive capability. In Comact, there are 2191 fragments of 1000 words. Therefore, according to the clustering model, on average a word with some probability $p$ would be expected to occur in

$$\mu = \left(1 - e^{-2.02 \cdot 10^{0.787 \cdot p}}\right) \cdot 2191 = \left(1 - e^{-464 \cdot p}\right) \cdot 2191$$

fragments. For example, if $p = 10^{-6}$ then $\mu = 1.02$. Now consider all of the $V$ words in Comact whose occurrence probability is near $p$; continuing the example, there
are $V = 282$ words in Comact with $p$ near to $10^{-6}$. Some of these words occur in none of the 1000-word fragments generated, some in one, some in two, and so on. Poisson methods can be used to estimate how many of the $V$ words will occur in exactly $v$ fragments using the formula

$$V \cdot P(Y = v) = V \cdot e^{-\mu} \cdot \frac{\mu^v}{v!}$$

These estimates can be compared with the number of words that occur in exactly $v$ fragments. Graphs of these estimates for $p = 10^{-6}$ ($V = 282$) and $p = 10^{-5}$ ($V = 217$) are shown in Figures 5.7 and 5.8 (continuous line). Observed values are shown as points in these graphs. For comparison, we have also plotted estimates yielded by the Poisson approximation for word distribution (dashed line), for which $\mu$ is given by

$$\mu = (1 - e^{-1000p}) \cdot 2191$$

As can be seen, for these probabilities the clustering model gives a fair prediction, for a set of words of a given probability, how many fragments each of the words will occur in; the Poisson model estimates these values very badly. Similarly good results are given for $n = 100$ and $n = 10000$. Unfortunately, for probabilities greater than $10^{-5}$ there are not enough data points to use this method to verify the model.

These results also explain the wide scattering of $k$ values at low probabilities shown in Figure 5.5. This scattering is a consequence of the fact that different words of a given probability will occur in different numbers of documents, in accordance with the Poisson distribution discussed above. Hence this technique can be used to give bounds to the number of fragments likely to contain a given word. For example, as can be seen in Figure 5.8, 95% of the words of $p \approx 10^{-5}$ occur in at least 3 and at most 20 of the 2191 fragments of 1000 words.

Another way to verify the clustering model is to use it to predict the number of distinct terms in a document of a given length. Consider a collection of $N$ documents each of length $n$. Each word $w$ will occur in $p_n(w) \cdot N$ documents. Thus, across the collection, there will be

$$\sum_{w \in W} p_n(w) \cdot N$$
Figure 5.7: Distribution of $p_n$ values for $p = 10^{-6}$ and $n = 1000$ on Comact
Figure 5.8: Distribution of $p_n$ values for $p = 10^{-5}$ and $n = 1000$ on Comact
distinct word-document occurrences, where $W$ is the set of distinct terms in the collection. Thus each document should have

$$W(N) = \sum_{w \in W} p_n(w)$$

distinct terms.

If $p_n$ values are unknown, the Poisson approximation or the clustering model can be used to estimate these values, based on the $p$ values of the words in the document collection. In Figure 5.9 we graph the number of distinct words in the actual documents of Comact, and also graph the clustering and Poisson estimates of $W(N)$ using $clus_n$ and $pois_n$ respectively to estimate $p_n$. As can be seen, the clustering model gives a good estimate of the number of distinct words in real documents.

If $p$ values are also unknown, they can be estimated by measures such as the Zipf distribution [154, 155] or amendments to the Zipf distribution such as that proposed by Mandelbrot [86]. We describe such a technique in Chapter 6, however, this additional level of approximation increases the degree of error in the approximation.

### 5.6 Limitations of the clustering model

An obvious limitation of the clustering model is that, unlike the Poisson approximation, the parameters of the model vary between collections. However, this limitation also applies to other models of text, such as the Zipf distribution. Furthermore, it is possible to choose typical $\alpha$ and $\beta$ values, and base estimates on these. Based on the data given in this chapter, it is possible to choose values of $\alpha$ and $\beta$ that are almost certain to exceed the actual values, yet would give values for $clus_n$ that are significantly less than $pois_n$. For example, if $\alpha$ is 60 and $\beta$ is 0.9, then $clus_n < pois_n$ for all $n \geq 60$. If more accurate values for $\alpha$ and $\beta$ are required they can be estimated using the method described in Section 5.4. We found that accurate estimates could be made using only 5% of the data sampling from every document.

A more serious problem is that the model is not always a close fit to observed data. Although it is almost always closer than the Poisson approximation, the model
Figure 5.9: Number of distinct terms in each document of Comact
is not very accurate for large $p$ values. This inaccuracy is particularly noticeable for large $n$. This divergence arises because we have assumed that $k$ is a constant for a given document collection and fragment size. In fact, as can be seen in Figure 5.5, $k$ drops as $p_n$ approaches 1. This effect is almost nonexistent for smaller $n$, but becomes evident (for Comact) for $n > 5000$. The most straightforward solution to this problem is to consider alternative forms of $\psi$. One form of $\psi$ that seemed promising was

$$\psi(n, p) = 1 - e^{-\gamma p^\sigma}$$

where both $\gamma$ and $\sigma$ are dependent on $n$. Thus $clus_n$ would be defined by

$$clus_n = 1 - e^{-(1-e^{-\gamma p^\sigma})}$$

For any given $n$, values for $\gamma$ and $\sigma$ can be found by fitting curves to graphs of $p$ against $-\log_e(1 - p_n)$. The resulting $clus_n$ is a very good fit to the observed data for all $p$ values. However, we could not identify any relationship between $n$ and these parameters.

Another limitation is that the model only applies to a range of sizes within each collection. For example, in Comact $\alpha$ and $\beta$ were computed by examining fragments between 60 and 40000 words in length. However, $clus_n$ with these parameters does not provide as good a fit to the data for fragments of other lengths as it does for lengths in this range.

### 5.7 Summary

We have shown that existing techniques do not give a good estimate of the probability that a document of a given length contains a word of a given probability. We have proposed a new measure that allows for the tendency of words to cluster, the clustering model, and have shown that this model gives a much better estimate of the probability that the document contains the word than does the Poisson approximation. The parameters of this model vary between document collections, and indicate the degree to which words cluster in a collection. We have also shown
that the clustering model can be used to give bounds to the number of documents likely to contain a given word, and to estimate the number of distinct words in a document.
Chapter 6

Physical Schemas

Conventional relational database systems are designed to support retrieval of information that has simple, repetitive structure. Documents, however, can be large and have a hierarchical physical structure; long documents usually contain several sections, each of which may contain several subsections or other logical units such as paragraphs or tables. Such structures can be difficult to efficiently store and retrieve in conventional relational database systems. The approach we have used in this thesis is to store documents at the physical level in nested or non-first-normal-form relational database systems. In this chapter we discuss how physical schemas for document storage and retrieval can be efficiently designed for a nested relational database system with signature file indexing, and give a detailed analysis of the space requirements and retrieval times of different document schemas for Boolean retrieval in such a database system.

A detailed analysis requires an understanding of the underlying indexing scheme; we assume the signature file techniques described in Section 6.1. An alternative to assuming signature file indexing schemes would be to assume compressed inverted file indexes [156]; this would not significantly effect the relative costs of retrieving data from the different physical schemas.

Using a nested relational database system a document can be represented as a single tuple in which the set of sections is a nested table and each section contains a nested table of subsections. We proposed an alternative view in Chapter 1 where we
suggested that documents should be broken into fragments, blocks of text holding logical units such as a paragraph. Because fragments are small, they are cheap to retrieve from disc, and signature file index sizes can be contained. Use of fragments can increase the size of databases, however, and makes some kinds of queries more expensive to evaluate.

We consider in detail the relative costs of some fragmented and unfragmented schemas for storing documents, and give formulas by which space and time requirements can be assessed. We present, in Section 6.2, an unfragmented schema and two fragmented schemas for storing documents in nested relations. We analyse the space requirements of each of these schemas in Section 6.3, and in Section 6.4 we discuss the types of queries we might expect to encounter. In Section 6.5 we discuss costs of these types of queries for each database schema, and we estimate optimal fragment sizes for an example document collection. We discuss the outcome of our comparison in the final section.

6.1 Signature file index schemes

Signature file index schemes can be used to index words and word pairs occurring in the body of a document, permitting queries on document content [67, 116]. For relations with large numbers of tuples, Roberts [108] proposed a bit-sliced implementation that should be used to reduce index look-up costs. Signature file indexes consist of a signature for each tuple (or record) in a table to be indexed; the length of each signature is proportional to the largest number of distinct words in a record. In this section, we give a summary of a generalized signature file architecture, as implemented in Atlas and described by Sacks-Davis et al. [117]; this includes an overview of the multiple organizational scheme of Kent et al. [67] we will be using throughout this chapter.

All signature file indexing schemes convert term values into signatures.
term signature

A term signature is an encoding of a term’s value. Each signature is of a fixed-length, \( b_r \) bits. The value of an indexed term is used to generate a series of hash values between 0 and \( b_r - 1 \), and the bits in the signature in the corresponding positions are set to one. All other bits are set to zero.

With superimposed coding index methods a distinct signature called the record descriptor is associated with each tuple.

record descriptor

A record descriptor is an encoding of the term values used to index the record; the terms can come from more than one attribute, and each attribute can contain many terms. A record descriptor is obtained by superimposing (using a bitwise OR function) the term signatures of all the terms contained in the record.

A distinct query descriptor is formed by the same technique.

query descriptor

A query descriptor is obtained by superimposing (using a bitwise OR function) the term signatures of all the terms contained in a conjunctive query.

To test whether a query term is contained in a given record, the query descriptor is compared with the record descriptor. This process can result in false matches, where the index cannot distinguish some records that do not match from those records that do match.

The simplest way to implement signature files is using bit strings.

bit string

In a bit string implementation, signatures are stored as a sequence of bits, together with a pointer to the corresponding data record.

At query time, each signature is retrieved and examined; if a signature matches a query, the corresponding data record is fetched. The bit string implementation
becomes inefficient for large tables since the whole signature file must always be retrieved.

An alternative to the bit string implementation is bit slicing.

**bit slicing**

Signature files are two dimensional bit arrays; storing the array by column rather than by row results in bit slicing.

Queries can be made more efficient by bit slicing because, with this technique, only those bits in each record descriptor in positions where bits are set in the query descriptor need to be retrieved at query time. This technique reduces query times at the expense of increased insertion costs, since with bit slices the bits of a single record descriptor are not stored contiguously and cannot be retrieved in a single disk access.

Further gains in efficiency can be obtained if signatures are formed for blocks of records as well as for single records. This results in a two level implementation of signature files, in contrast to the one level scheme described above. With this approach a table of $R$ records is considered as consisting of $\frac{R}{B}$ blocks, each of $B$ records.

**block descriptor**

A *block descriptor*, of length $b_s$, is formed from all of the terms in the records belonging to that block. As a consequence $b_s$ (for the terms in all the records in a block) will in general be much greater than $b_r$ (for the terms in a single record).

The block descriptors are stored in a separate file from the file of record descriptors.

At query time, the block descriptors are first examined and only those record descriptors in matching blocks are retrieved. The block descriptors are stored in bit slice form so the query costs are independent of the block descriptor length, $b_s$. Compared to the one level implementations, the length of a bit slice is $\frac{R}{B}$ rather than $R$. Since $\frac{R}{B}$ is an order of magnitude smaller than $R$, with a two level implementation single bit slices can be retrieved in a single disk access even for tables with millions
of records. As a consequence, these methods provide efficient query times for very large tables.

With only a single block descriptor file, however, false block matches can occur for queries that specify two or more terms, when these terms are contained in different records within the block but not in any single record. This type of false block match is avoided in a two level signature file implementation referred to as the \textit{multi-organizational} scheme.

\textbf{multi-organizational scheme}

If $K$ bits are set per term in the two level scheme with a single block descriptor file, then with the \textit{multi-organizational scheme}, $K$ block descriptor files are stored and in each block descriptor file a single bit is set per term. In the multi-organizational scheme, the way in which records are grouped to form blocks of records can differ from block descriptor file to block descriptor file, hence the term \textit{multi-organizational}.

No extra storage overhead is generated compared to the conventional two level implementation.

With this approach, it is possible to determine whether a record matches a query by examination of the block descriptors. Since each record is now associated with a unique combination of block numbers, a record descriptor file is no longer required.

Multi-organizational indexes use less space than the simple two level indexes. For a typical Atlas database, a simple two-level index would use 35\% to 45\% of the original data space, whereas a multi-organizational index would use 25\% to 35\% of the original data space. The exact space requirement depends on the proportion of distinct words per record to overall words per record, and on how words are indexed: for example, an index on stems and exact match would be about 25\% larger than an index on stems alone.

The choice of which signature file method to use for a particular table to be indexed will depend on a number of factors such as the size of the table being indexed and the requirement for fast query processing as opposed to fast update. The single level bit string implementation is suitable for tables with a relatively
small number of records and supports efficient update as well as query processing. On the other hand, the multi-organizational scheme provides fast query processing for very large files but interactive update is expensive.

A generalized signature file architecture, as implemented in Atlas, is shown in Figure 6.1 (based on a similar figure in Sacks-Davis et al. [117]). It is characterized by the length of the block descriptors $b_s$, the length of the record descriptors $b_r$, the number of block descriptor files $K$, and the block size $B$. Choosing $K = 0$ and $b_s = 0$ results in the bit string implementation; choosing $K > 0$ and $b_s > 0$ gives bit slicing; note that these values must be both zero or both non-zero. The one level bit slice implementation is derived by choosing $b_r = 0$ and $B = 1$. Choosing $B > 1$ yields bit slicing with blocking; if $b_r > 0$ the two level scheme is derived, and if $b_r = 0$, the multi-organizational scheme is derived, which does not require record descriptors. Note that the functions that map records to blocks are different for the two level and multi-organizational schemes.

Practical implementations of signature files contain other features for efficient
processing of large data files. For commonly occurring terms, dedicated bit slices can be reserved in the signature file. This can result in more efficient query processing, due to a reduction in false matches. A vocabulary of common terms is maintained in main memory. The number of commonly occurring terms chosen will depend on the signature file method chosen and the amount of main memory available for the vocabulary, and can vary from a few terms to tens of thousands of terms.

Faloutsos [49] compares signature methods with other access methods for text, including multi-attribute hashing and inverted files. In 1985 he concluded that “the signature file approach seems to be the most promising for archiving documents” [49]; however, Zobel et al. [156] suggest this may no longer be the case given recent advances in inverted file management.

6.2 Storing documents in nested relations

Documents might be stored in a nested relational database as follows: each document could be represented as a single tuple in which the set of sections is a nested table and each section contains a nested table of subsections. In such a schema, however, entire documents must be retrieved in response to queries, because the unit of retrieval is a tuple. Moreover, queries on more than one word can match documents in which those words are widely separated and are probably unrelated. Thus some queries will lead to large amounts of irrelevant material being retrieved. Most importantly, because of the range of document sizes that can occur in a large document collection, bit-sliced signature file indexes can become unacceptably large or unacceptably dense resulting in a high rate of false matches, making this approach to document storage impracticable.

As an alternative, documents can be broken into fragments. There are several advantages to using fragments (or nodes) rather than whole documents. First, they are of a more manageable size than whole documents; size variation between fragments can be constrained to be far less than the size variation between documents, thus minimizing the size of signature files. Second, if a user looks for tuples containing a set of words, there is some guarantee that the words occur close together in the
retrieved text. Third, use of fragments reduces the volume of disc traffic; retrieving a fragment is considerably cheaper than retrieving an entire document. Fourth, in many applications it is natural to consider documents as consisting of parts rather than as a whole; for example, the nodes in a hypertext system.

One disadvantage of fragmenting documents is that it can become more expensive to find information about the document from which a given fragment was drawn. It may therefore be useful to associate title information (document title, author name, and so on) with each fragment. If title information is usually retrieved with each fragment then retrieval is more efficient if we store the title information with each fragment. If fragmented documents are to be stored in a minimum of space, however, title information should not be repeated. The simplest way to effect this is to store the text in one table and title information in another. To allow a fragment to be joined to its header, and to allow documents to be reconstructed, reference attributes could be stored with each fragment and each title.

In this chapter we use a simpler document database schema than presented in earlier chapters, a document database containing just documents and fragments (nodes of a uniform size without any hypertext links). Our analysis considers three organizations: monolithic, segmented, and duplex. These organizations are analogous to the nested relational schemas in Figures 4.2, 4.4, and 4.8.

**monolithic schema**

In the *monolithic schema* each document is represented by a single tuple containing a nested table of fragments called nodes.

This schema is illustrated in Figure 6.2. In terms of time and space requirements, this schema is similar to having no fragmentation.

**segmented schema**

In the *segmented schema* each document is represented by a number of tuples, each containing title and author information.

This schema is illustrated in Figure 6.3; title and author information is repeated but the use of reference attributes is avoided.
duplex schema

In the *duplex schema* each document is represented by a header tuple containing title and author information, and a nested table of foreign keys of fragments containing the text of that document.

This schema is shown in Figure 6.4. Reference attributes (foreign keys) have been introduced to facilitate access between fragments and titles. The separation of text and other information in the duplex schema means that the structure of documents can be explored without any text having to be retrieved; compare this with the monolithic schema, where document structure is embedded. We refer to databases with a segmented or duplex schema as *fragmented*.

Given the integrity constraints shown the monolithic, segmented and duplex schemas are mapping equivalent.

These approaches to document management have been developed as a consequence of experience with document databases and Atlas. Methods for arriving at optimal nested relational designs have also been considered by Hafez and Ozsoyooglu [60] and Wen [150]. For example, once a fragment size had been chosen, these
fragment [

document (  
    docid INTEGER,  
    title TEXT,  
    authors [  
        firstname TEXT,  
        surname TEXT  
    ]  
  ),  
  fragid INTEGER,  
  data TEXT  
] KEY = (fragid)

Template dependency

document [ docid title authors[*] fragid data]

hypothesis

<table>
<thead>
<tr>
<th>X1</th>
<th>Y1</th>
<th>Z1</th>
<th>V1</th>
<th>W1</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>Y2</td>
<td>Z2</td>
<td>V2</td>
<td>W2</td>
</tr>
</tbody>
</table>

conclusion

| Y1=Y2 | Z1=Z2 |

Figure 6.3: A schema for segmented storage of documents
Figure 6.4: A schema for duplex storage of documents
methods could be used to derive the duplex structure given the monolithic structure and a series of sample queries. However, these methods cannot be used to derive information about fragment sizes, nor can they be used to identify cases in which information should be repeated, such as in segmented schemas.

Another database system designed for document management is MULTOS [16]. In contrast to our fragmented schemas, documents are stored as single entities and the underlying storage organization is not based on the nested relational model. In MINOS [29], which is also designed for document management, documents are represented as complex objects with explicit structure. However, like MULTOS, MINOS stores documents in a monolithic structure. Mistral/11 is an early document retrieval system, and was based on the relational model [83]. Mistral/11 also uses monolithic structures.

6.3 Space analysis

In this section we compare the space requirements of the schemas described in Section 6.2. We assume that words are distributed in text as given by the clustering model described in Chapter 5. In this model, the probability that a document or fragment of \( n \) words contains a given word \( w \) with occurrence probability \( p(w) \) is given by \( p_n(w) \) as follows.

\[
p_n(w) = 1 - e^{-\alpha^{1-\beta} \cdot n^{\beta} \cdot p(w)}
\]

The parameters \( \alpha \) and \( \beta \) are dependent on nature of the particular document collection; typical values are 2.74 and 0.920 respectively. These values are derived from the King James version of the Bible, and we will assume them throughout this chapter.

Initial investigations extending this model (conducted by Justin Zobel but not yet reported) indicate that the probability that a document or fragment of \( n \) words contains all of the words \( w_1, \ldots, w_m \) can be approximated by the product of the
\( p_n(w_i) \)’s

\[
p_n(w_1, \ldots, w_m) = \prod_{i=1}^{m} p_n(w_i)
\]

where the words \( w_1, \ldots, w_m \) are independent.

Index sizes depend on the number of distinct words. An approximation to the number of distinct words in a document or fragment of \( n \) words is given by

\[
W(n) = \sum_{i=1}^{W} p_n(w_i)
\]

where \( W \) is the number of distinct words in the database and \( p_n(w_i) \), the probability of the \( i \)th-ranked word, is assumed to follow the Zipf distribution

\[
p_n(w_i) = \frac{1}{(\log_e W + \gamma).i}
\]

in which \( \gamma = 0.5772 \) is the Euler-Mascheroni constant [151]. Throughout this chapter we assume that \( W \) is 500,000. For the reader’s reference, some typical values of \( W(n) \) are shown in Table 6.1.

<table>
<thead>
<tr>
<th>( n )</th>
<th>50</th>
<th>100</th>
<th>500</th>
<th>1,000</th>
<th>5,000</th>
<th>10,000</th>
<th>50,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W )</td>
<td>50,000</td>
<td>34.3</td>
<td>61.1</td>
<td>227</td>
<td>396</td>
<td>1390</td>
<td>2330</td>
<td>7330</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>34.7</td>
<td>62.2</td>
<td>235</td>
<td>412</td>
<td>1480</td>
<td>2520</td>
<td>8300</td>
</tr>
<tr>
<td></td>
<td>500,000</td>
<td>35.6</td>
<td>64.3</td>
<td>249</td>
<td>442</td>
<td>1650</td>
<td>2880</td>
<td>10200</td>
</tr>
</tbody>
</table>

Table 6.1: Estimated number of distinct words \( W(n) \) in fragment of size \( n \)

Document databases can be described by several parameters.

- The number of documents stored is denoted by \( N \), assumed to be 100,000, the number of words of title and author information is \( N_t \), assumed to be 50, and the average number of words in the text of each document is \( N_w \), assumed to be 10,000. Each word is \( N_b \) bits long, assumed to be 50. This implies that approximately 6,300 megabytes of data is to be stored.
- In segmented and duplex databases, documents are divided into $F$ fragments. Each fragment contains on average $N_{fw} = N_w/F$ words.

- Multi-organizational bit-sliced signature file methods are used to index data and all fields (other than foreign keys) are indexed. The number of index words in a relation is the average number of distinct words in each tuple times the number of tuples in the relation. The size in bits of the signature file for a relation is $S$ times the number of index words in the relation, where $S$ is assumed to be 32.

- Reference attributes (foreign keys) occupy $S_k$ bits, assumed to be 32.

In such a collection $p(w)$ would typically range from about $10^{-8}$, for words that only occur a few times (note that Zipf’s formula substantially overestimates this value), to $5 \times 10^{-2}$, for words such as the. Query words tend to be less common, so we assume that the most common query words would have $p(w) \simeq 10^{-4}$.

Approximate space requirements are given by the following formulas.

<table>
<thead>
<tr>
<th>Tuple size (bits)</th>
<th>Data (bits)</th>
<th>Index (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Monolithic</strong></td>
<td>$S_m = (N_t + N_w).N_b$</td>
<td>$S_m.N$</td>
</tr>
<tr>
<td><strong>Segmented</strong></td>
<td>$S_s = (N_t + N_{fw}).N_b$</td>
<td>$S_s.N.F$</td>
</tr>
<tr>
<td><strong>Duplex</strong></td>
<td>$S_t = N_t.N_b + S_k.F$</td>
<td>$S_t.N$</td>
</tr>
<tr>
<td><strong>document</strong></td>
<td>$S_f = N_{fw}.N_b + 2.S_k$</td>
<td>$S_f.F.N$</td>
</tr>
</tbody>
</table>

Note that these approximations assume that each document is of roughly the same size. A more accurate estimate of the size of indexes would be based on the largest number of distinct words in a tuple in a relation rather than on the average number of distinct words, so that the formula given above will tend to underestimate the size of indexes, and in particular the size of indexes of monolithic databases. For example, if documents ranged up to 100,000 words in length with an average length of 10,000 words, the size of the monolithic index would be about six times greater
than that given by the above formula since signatures would need to be based on
the size of the largest documents to minimize false matches.

In Figure 6.5 we show how the estimated sizes of indexes and databases varies as
fragment size varies. Segmented databases are the largest, but, not surprisingly, the
difference decreases as the size of fragments grows. One of the main differences in the
space requirements is in the size of the indexes: the fragmented structures, with their
greater number of index words, have larger indexes than the monolithic structure.
In arriving at our estimates we optimistically assume that each document in the
collection is of the same size, an assumption that favours the monolithic structure.

Space requirements can be reduced if data is compressed [89, 152]. However, text
compression does not significantly affect index size: it reduces \( N_b \) only, to perhaps
15 in our example. We do not show results for compressed databases in this chapter,
but compression favours monolithic structuring in space and fragmented structuring
in time.

### 6.4 Types of queries

There are many ways in which users might search for documents in a document
database system. For example, users are as likely to request documents pertaining
to a subject or set of words as to request the document with a given title and set of
authors. In Chapter 1 we focussed on three types of queries: Boolean queries, queries
involving similarity matching and ranking, and exploration or browsing queries. Ex-
ploration or browsing queries are supported by the hypertext structures discussed in
earlier chapters. Queries involving similarity matching and ranking require special-
ized indexes containing additional information, such as term weights, and we shall
not consider them further. In our analysis in this chapter, we restrict our attention
to Boolean queries. We focus on conjunctive queries as queries involving disjuncts
can be decomposed into a sequence of queries containing only conjuncts.

We analyse three ways in which documents might be accessed: \textit{query by title},
\textit{query by fragment content}, and \textit{query by document content}. 
Figure 6.5: Predicted sizes of indexes and databases against fragment size
query by title

Users *query by title* when they search for title information on the basis of words occurring in a title or other header fields such as authors names.

An example of a query by title is the query requesting ‘find the titles of all documents containing the words *computable* and *numbers* in the title’.

query by fragment content

Users *query by fragment content* when they request the system to return the appropriate parts of documents that match a query rather than returning entire documents that match. In this strategy, monolithic documents which contain all of the query words, but in which the words are widely separated are not returned.

We believe that query by fragment content will be the most common: for example, users who request text containing the words *computable* and *numbers* are likely to only be interested in documents in which these words are in the same sentence or paragraph.

query by document content

Users *query by document content* when they search for a document on the basis of words occurring in the document, wherever those words appear.

An example of a query by document content is the query requesting ‘find all documents containing the words *computable* and *numbers*’.

Each of these kinds of query can be difficult to support in a key-based relational database system. Words and combinations of words are not keys in the conventional sense. Moreover, if fragments are used, the database system must be able to find fragments in the order in which they occurred in the original document; in conventional relational systems, however, tuples are unordered.
6.5 Query costs on fragmented and monolithic databases

In this section we compare costs of queries on monolithic, segmented, and duplex databases. Query costs for fragmented databases are different from query costs for monolithic databases. In monolithic databases each retrieved tuple is large, and in general more irrelevant tuples are retrieved since the query words may not occur close together in the retrieved text. On the other hand, in fragmented databases each retrieved tuple is small, and fewer irrelevant tuples are retrieved. However, some queries on fragmented databases will have join costs that would not exist in monolithic databases.

We now consider the costs of typical queries to a database with the structures described in Section 6.2. In this section, we estimate the costs of conjunctive queries where we wish to retrieve those documents or fragments containing all of the words \( w_1, \ldots, w_m \), where \( m \geq 1 \). In addition to the assumptions made in Section 6.3, we assume the following.

- The unit of retrieval is a tuple.

- Monolithic documents are stored contiguously on disc, and the fragments of a duplex document are stored contiguously on disc. The latter assumption minimizes seek times when several fragments are retrieved from the one document. We denote seek+latency time by \( T_s \), and assume an average of \( 10^{-2} \) seconds.

- There is a cost associated with retrieving and processing each bit of data. We denote this cost by \( T_d \), and assume \( 10^{-6} \) seconds per bit. Processing costs include checking that retrieved tuples are valid (signature methods can return false matches) and processing text into a format appropriate for display. Similarly, we assume that there is a cost \( T_i \) of retrieving and processing each bit of index, and assume \( 10^{-7} \) seconds per bit.

- Each word sets \( K \) bits in the signature of the tuple containing that word,
where $K$ is assumed to be 8, and signatures are formed for blocks of tuples rather than individual tuples. Block size is $B$, assumed to be 32.

- The cost of looking up signature file for a single term query on a relation of $R$ tuples is $K.(T_s + T_i.R/B)$.

- The cost of looking up signature file for a multiple term query where $m$ is much less than $K$ (say $m \leq \frac{K}{2}$) is the same as for a single term query. For multiple term queries where $m$ is greater than $K$ the cost is approximately $m.(T_s + T_i.R/B)$. In the following we assume $m$ is much less than $K$.

### 6.5.1 Query by title

Consider queries by title information such as title or author. Each tuple in the document relation of a duplex database holds all of the title information held in each tuple in the equivalent monolithic schema, but tuples in the document relation in the duplex schema (which consist of title information and foreign keys of fragments) are much smaller. For segmented databases, we assume that a bitmap (of size $N.F$ bits) is used to indicate whether each tuple is the first tuple of a document, so that other tuples of the document can be ignored. Approximate costs for query by title are given by the following formulas.
**Monolithic**

Time to look up index \( I = K_s(T_s + T_i \cdot \frac{N}{B}) \)

Number of matching tuples \( M = p_{N_i}(w_1, \ldots, w_m).N \)

Total time \( I + T_s.M + T_d.S_m.M \)

**Segmented**

Time to look up index \( I = K_s(T_s + T_i \cdot \frac{N}{B}) \)

Number of matching tuples \( M = p_{N_i}(w_1, \ldots, w_m).N \)

Total time \( I + T_s.M + T_d.S_m.M \)

**Duplex**

Time to look up index \( I = K_s(T_s + T_i \cdot \frac{N}{B}) \)

Number of matching tuples \( M = p_{N_i}(w_1, \ldots, w_m).N \)

Total time \( I + T_s.M + T_d.S_m.M \)

These estimates of retrieval times, for single word queries, are illustrated in Figure 6.6; the top graph shows how retrieval time varies with fragment size for \( p(w) = 10^{-6} \), and the bottom graph shows how retrieval time varies with probability for \( N_{fw} = 100 \). The number of tuples retrieved is the same for each schema, and is therefore not shown. As can be seen, for almost all probabilities and fragment sizes queries to the duplex database are faster than to the segmented structure, and much faster than to the monolithic structure. The only exception to this is for very low \( p(w) \), in which case no tuples are retrieved at all and the small costs are due to the indexing.

### 6.5.2 Query by fragment content

We believe that in document database systems, in many applications the most common kind of query will be to find documents on the basis of content. We first consider the costs of querying by fragment content. Note that some documents that satisfy a query will contain no fragments that satisfy the query, because the query words are widely separated in the document. Also, because fragments are stored contiguously, once a fragment of a document has been retrieved, no seek is required.
Figure 6.6: Single-word queries by title against fragment size and against word probability
for subsequent fragments from the same document. Thus, in fragmented databases, the number of seeks is at most the number of documents involved in the query, which will be at most the number of documents retrieved from the monolithic database (hence the use of the function min in the following formulas).

Approximate costs for queries by fragment content are as follows.

**Monolithic**

- Time to look up index: $I = K(T_s + T_i \frac{N}{B})$
- Number of matching tuples: $M = p_{NW}(w_1, \ldots, w_m) . N$
- Total time: $I + T_s . M + T_d . S_m . M$

**Segmented**

- Time to look up index: $I = K(T_s + T_i \frac{N.F}{B})$
- Number of matching tuples: $M = p_{N_fw}(w_1, \ldots, w_m) . N . F$
- Total time: $I + T_s . \min(M, p_{NW}(w_1, \ldots, w_m) . N) + T_d . S_m . M$

**Duplex**

- Time to look up index: $I = K(T_s + T_i \frac{N.F}{B})$
- Number of matching tuples: $M = p_{N_fw}(w_1, \ldots, w_m) . N . F$
- Total time: $I + T_s . \min(M, p_{NW}(w_1, \ldots, w_m) . N) + T_d . S_f . M$

In a variant of this type of query, title information as well as the retrieved text is displayed to the user, providing contextual information about the text. In monolithic and segmented structures, this information is retrieved in any case, but extra operations are needed in duplex schemas, to retrieve the tuples of title information. The additional costs are as follows.

**Duplex — variant query — additional costs**

- Time to look up index: $I = K(T_s + T_i \frac{N}{B})$
- Number of matching tuples: $M = \min(p_{NW}(w_1, \ldots, w_m) . N, p_{N_fw}(w_1, \ldots, w_m) . N . F)$
- Total additional time: $I + T_s . M + T_d . S_t . M$
Times and numbers of matching tuples for queries by fragment content, for single word queries on words with occurrence probability $p(w) = 10^{-7}$, are shown in Figure 6.7; results for single word queries on words with $p(w) = 10^{-6}$ are shown in Figure 6.8; and results for single word queries on words with $p(w) = 10^{-5}$ are shown in Figure 6.9. Times and numbers of matching tuples for queries by fragment content, for multi-word queries on two words each with $p(w) = 10^{-4}$, are shown in Figure 6.10. Times and numbers of matching tuples for queries by fragment content, for multi-word queries on three words each with $p(w) = 10^{-4}$, are shown in Figure 6.11. Again, it can be seen that (non-variant) queries on the duplex database are cheaper than queries to the other structures, and are much cheaper than queries to the monolithic structure. As the number of query words increases, the relative cost of using the monolithic structure increases drastically, as illustrated by the 10,000-fold difference in times between the fragmented and monolithic schemas in Figure 6.11. For queries with a larger number of words, the difference is even greater.

In the top graph in Figure 6.7, it can be seen that, in duplex databases, there exists a fragment size that has the minimum retrieval time for queries on words with probability $10^{-7}$. This minimum depends on the probability of the query word: for example, as can be seen in Figure 6.8, for query words with occurrence probability of $10^{-6}$, the optimal fragment size is smaller. The minimum retrieval time occurs for fragments that are relatively small, somewhere in the region of a paragraph of text. Note that reducing fragment size down to (say) 100 words makes queries to segmented structures cheaper to evaluate, but, as is shown in Section 6.3, at a considerable space penalty.

If the variant type of queries requesting title information are expected to be common, then if the disc space is available the segmented schema is preferable to the duplex as can be seen from Figures 6.7, 6.8, 6.9, 6.10, and 6.11.

Note that, as discussed in Section 6.4, some retrieved monolithic documents may have to be discarded because the query words are not near to each other in the text of those documents. In some cases, as illustrated in Figure 6.11, the number of documents to be discarded can become very large. Use of a secondary index that
Figure 6.7: Single-word queries by fragment content against fragment size for $p(w) = 10^{-7}$
Figure 6.8: Single-word queries by fragment content against fragment size for $p(w) = 10^{-6}$
Figure 6.9: Single-word queries by fragment content against fragment size for $p(w) = 10^{-5}$
Figure 6.10: Two-word queries by fragment content against fragment size for both $p(w) = 10^{-4}$
Figure 6.11: Three-word queries by fragment content against fragment size for each $p(w) = 10^{-4}$
allowed access to monolithic documents on the basis of fragments of the documents would eliminate this problem, at the cost of extra space to store the index. Even with this optimization, queries to the monolithic schema would still be slower than queries to the other schemas, because of the larger amount of data to be retrieved.

6.5.3 Query by document content

In monolithic databases, querying by document content is identical to querying by fragment content. In fragmented databases, complex evaluation strategies are required. One strategy is to retrieve all fragments that contain any of the query words, use them to determine which documents contain all of the query words, and then retrieve all of the information for that document. To retrieve a whole document, the locations of the first and last fragments must be found, as well as the title of the document in the duplex case. All of the data stored between the first and last fragments of a document should be retrieved, as documents are stored contiguously on disc. Approximate costs are as follows.

Monolithic As for query by fragment content.

Segmented

<table>
<thead>
<tr>
<th>Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to look up index</td>
<td>( I = K.m.(T_s + T_i.\frac{N.F}{B}) )</td>
</tr>
<tr>
<td></td>
<td>( +2.K.(T_s + T_i.\frac{N.F}{B}) )</td>
</tr>
<tr>
<td>Number of matching tuples</td>
<td>( M = \left(p_{N_f}^w(w_1) + \ldots + p_{N_f}^w(w_m)\right)N.F )</td>
</tr>
<tr>
<td></td>
<td>( +p_{N_w}(w_1, \ldots, w_m)N.F )</td>
</tr>
<tr>
<td>Total time</td>
<td>( I + T_s.\left((p_{N_f}^w(w_1) + \ldots + p_{N_f}^w(w_m))F \right) )</td>
</tr>
<tr>
<td></td>
<td>( +p_{N_w}(w_1, \ldots, w_m)N + T_d.S_s.M )</td>
</tr>
</tbody>
</table>
Duplex

Time to look up index

\[ I = K.m.(T_s + T_i \cdot \frac{N.F}{B}) + K.(T_s + T_i \cdot \frac{N}{B}) \]
\[ + 2.K.(T_s + T_i \cdot \frac{N.F}{B}) \]

Number of title tuples

\[ M_t = p_{N.w}(w_1, \ldots, w_m).N \]

Number of fragment tuples

\[ M_f = (p_{N_{f.w}}(w_1) + \ldots + p_{N_{f.w}}(w_m)).N.F \]
\[ + p_{N.w}(w_1, \ldots, w_m).N.F \]

Total time

\[ I + T_s.((p_{N_{f.w}}(w_1) + \ldots + p_{N_{f.w}}(w_m)).N.F \]
\[ + 2.M_t) + T_d.(S_t.M_t + S_f.M_f) \]

Costs given by these formulas are shown in Figure 6.12, for queries on three words each with \( p(w) = 10^{-4} \). As can be seen, this type of query is more expensive in fragmented databases, particularly segmented databases.

In an environment in which this kind of query is expected to be common, a monolithic structure would be superior. However, we believe that this kind of query would be rare. For example, in hypertext systems users deal almost exclusively with parts of documents, and would rarely or never query a whole document.

6.6 Summary

It is difficult to efficiently store and retrieve documents using conventional relational database systems. Documents have more complex structure than conventional data, and even documents of the same kind can vary greatly in organization and size. Moreover, the kinds of queries issued to document databases are different from those issued to conventional databases. We have shown that documents can be stored in nested relational database systems with bit-sliced signature file indexing, in which there is support for efficient access by content. Changing the indexing scheme would not alter the general conclusions of this chapter, for example, replacing the superimposed codeword index by a compressed inverted file index [156].

The major result in this chapter is that to minimize retrieval costs documents should be broken into fragments, each of which should be stored in a separate tuple. There are several reasons for this. First, in many applications most queries will be
Figure 6.12: Three-word queries by document content against fragment size for each $p(w) = 10^{-4}$
by title or fragment content; fragmentation permits much faster access to the data for such queries, because in general much less data is retrieved from a duplex or segmented database than from a monolithic database. Our results indicate that for large document collections queries to monolithic schemas are so slow that such schemas are impractical. Second, documents can vary greatly in length, which can cause bit-sliced signature file indexes to become unreasonably large. In fragmented databases, size variations are contained so that this problem does not arise. Third, fragments are similar to the units of text handled by some document database applications, for example hypertext.
Chapter 7

Conclusions and future directions

We have provided a framework for the design of document database systems that includes both a basis for modelling of document structure and a basis for efficient retrieval from very large collections of documents. Complex objects based on nested relations are a suitable data model for representing documents, and can be accessed and manipulated using TQL — a query language which has special text operators and allows access to complex objects via reference attributes. A logical schema which represents documents as complex objects should be designed to allow queries to be expressed in a simple manner, by modelling each type of entity (such as documents, nodes, and links) as a separate class of objects. Such a logical schema can be mapped to any one of several equivalent nested relational physical schemas; choosing the best physical schema is an optimization problem to minimize the cost of query processing. Comparing the cost of text based queries under different physical schemas requires a model of word clustering to provide an estimate of the probability that a fragment of text contains a given word.

In Chapter 1 we identified the main characteristics of document database systems. These included the nature of text, document structure, and different query paradigms. We introduced a hyperbase example (a hypertext database) to use throughout the thesis to illustrate various requirements of document database systems. This example is given in a paper that was presented at the First Australian Multi-Media Communications and Applications and Technology Workshop.
We compared, in Chapter 2, some of the diverse range of data models that have been proposed over the last thirty years. Using our hyperbase example we identified those features in existing data models that were effective for representing document databases. Complex object and semantic data models have desirable features for modelling document databases. Implementation data models, that provide complex objects, allow semantic models to be easily transformed into physical storages structures. Two data models based on nested relations were proposed. One, a record-based model, can be used to describe the physical storage structures. The other, a higher-level object-based model, can be used to describe the logical structure of the database.

In Chapter 3, we addressed query language support for complex objects and text. We described in detail the query language TQL, which has powerful support for text compared with other languages proposed for the nested relational model. The language provides support for non-first normal form relations including text attributes, tuples, nested tables, and implicit joins using references. TQL queries can be directly made on both the record-based data model and object-based data models. The TQL language is described in a paper that appeared in the *Australian Computer Journal* in 1991 [136]. The implementation of TQL, which included a simple optimizer for TQL, is described in a paper on the Atlas system that will appear in the *IEEE Transactions on Knowledge and Data Engineering* [117]. Some problems of query optimization for TQL and the nested relational model are still to be thoroughly investigated; particular issues are the complexity of join operators for nested tables and the efficient support of operators such as nest and unnest. Higher-level query languages, graphical user interfaces, designed specifically for querying document databases are still to be fully developed. A possible language for querying hypertext databases is described in a paper that was presented at the *Second International Conference on Database and Expert Systems Applications* [53] in 1991. Such a language, together with a user interface, could be implemented using TQL embedded in an application program.

The interaction between the logical and physical aspects of document design
were considered in Chapter 4. It is desirable that the logical design of complex objects be based on the entities on which queries be made. However, the optimal complex objects at the physical level may differ from the complex objects desired at the logical level. Therefore we provided a mapping from the complex object model at the logical level to the physical record-based data model of Atlas; this mapping is not unique. We also provided a mapping into our complex object model from entity-relationship diagrams. The distinction between the requirements for logical design and physical design of complex objects was made in a paper that was presented at the Third Australian Database Conference in 1992 [139].

The approach to design using record-based and object-based models is more general than just document database design and can be applied to any database system based on complex objects, including object-oriented database systems. One area for further work on the logical complex object model is to extend the logical schema to allow a wider range of semantic constructors (for example, a class hierarchy with inheritance), while preserving a simple and efficient mapping to the physical schema. Another area for further work is updates to such schemes: we conjecture that it should be possible to determine the effect of updates on the underlying schema because of constraints specified by inverse relationships.

The cost of a text-based query depends on the degree of word clustering. A new statistical model of word clustering was developed to more accurately estimate query costs for determining an optimal physical schema. This clustering model for distribution of words in text, which we described in Chapter 5, appeared in a paper in the Journal of the American Society for Information Science in 1992 [141]. We showed how this model could be used to estimate the probability that a document contains a word and estimate the number of distinct words in a document. A common model for the distribution of words in text is the Poisson approximation to the binomial distribution. However, this measure ignores the effect of clustering and significantly overestimates the probability that a document contains a word. We proposed a new model that allows for the tendency of words to cluster, the clustering model, and showed that this model gives a much better estimate of this probability. The parameters of this model vary between document collections, and
indicate the degree to which words cluster in a collection. The clustering model can be used to give bounds to the number of documents likely to contain a given word, and to estimate the number of distinct words in a document.

The clustering model has applications to other problems related to text; for example, the importance of query terms in information retrieval can be estimated using this model. There are some interesting open problems concerning the clustering model. One is to model occurrences of compound terms such as word pairs within documents. Another is to model occurrences of documents containing each member of a set of words. Both of these problems are relevant to the problems of text compression and full text retrieval from databases. Future work on the clustering model will be required to give a more extensive parameterization of clustering; for example, estimating bounds on performance requires a measure of the divergence of word occurrences from their average behaviour.

In Chapter 6 we examined the design of efficient storage structures for document databases. The formulas describing the efficiency of the different physical structures use the clustering model we developed. We showed that documents can be stored in nested relational database systems with bit-sliced signature file indexing, in which there is support for efficient access by content.

The most significant result concerning physical structures was that documents should be broken into fragments, each of which should be stored in a separate tuple. Fragments are similar to the units of text handled by some document database applications, for example hypertext. We described two possible fragmented schemas, a segmented schema and a duplex schema. For both of these schemes, we analysed the relationship between fragment size, database size, and query response time. There is no fixed optimal fragment size, but, for queries that retrieve a small proportion of the database, smaller fragments give better retrieval time. However, a good fragment size would be such that each fragment contains a sentence or paragraph. Of the two schemas for fragmented databases, the duplex schema occupies substantially less space, and is faster for most of the query classes considered. Choice of schema will depend on the application, but we expect that duplex schemas will generally be preferred. These results are reported in a paper that was presented at the
Seventeenth International Conference on Very Large Data Bases in 1991 [157].

There are many variants of these physical schemes that might be considered: use of secondary indexes to provide access to fragmented databases on the basis of the content of whole documents rather than the content of fragments; merging the segmented and duplex schemas to get the best possible retrieval speed; and considering the costs of further query types. Further work could be done on the relationship between SGML markup and the generation of fragments. However, such investigations would not alter the result that fragmentation permits much faster access to data stored in large document databases.

The framework for the design of document databases systems presented in this thesis is a step towards a future in which document databases will replace the printed word as the primary medium for the storage and retrieval of information.
Bibliography


BIBLIOGRAPHY


## Glossary of notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
<th>Value used in Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )</td>
<td>fragment size above which clustering takes effect</td>
<td></td>
</tr>
<tr>
<td>( \beta )</td>
<td>degree of clustering</td>
<td></td>
</tr>
<tr>
<td>( \beta )</td>
<td>degree of clustering</td>
<td></td>
</tr>
<tr>
<td>( \beta_0, \beta_1 )</td>
<td>values in true regression line ( y = \beta_0 + \beta_1 \cdot x )</td>
<td></td>
</tr>
<tr>
<td>( \hat{\beta}_0, \hat{\beta}_1 )</td>
<td>point estimates for ( \beta_0, \beta_1 )</td>
<td></td>
</tr>
<tr>
<td>( \gamma )</td>
<td>Euler-Mascheroni constant</td>
<td>0.5772</td>
</tr>
<tr>
<td>( b_r )</td>
<td>signature length in bits</td>
<td></td>
</tr>
<tr>
<td>( b_s )</td>
<td>block descriptor length in bits</td>
<td></td>
</tr>
<tr>
<td>( B )</td>
<td>number of records per block</td>
<td>32</td>
</tr>
<tr>
<td>( \clus_n )</td>
<td>cluster model approximation for ( p_n )</td>
<td></td>
</tr>
<tr>
<td>( F )</td>
<td>number of fragments per document</td>
<td></td>
</tr>
<tr>
<td>( I )</td>
<td>seconds to search index</td>
<td></td>
</tr>
<tr>
<td>( k )</td>
<td>constant ( -\log_e(1-p_n)/p )</td>
<td></td>
</tr>
<tr>
<td>( K )</td>
<td>number of block descriptor files</td>
<td>8</td>
</tr>
<tr>
<td>( m )</td>
<td>number of query words</td>
<td></td>
</tr>
<tr>
<td>( M )</td>
<td>number of matching tuples</td>
<td></td>
</tr>
<tr>
<td>( n )</td>
<td>number of words in a fragment</td>
<td></td>
</tr>
<tr>
<td>( N )</td>
<td>number of documents</td>
<td>100,000</td>
</tr>
<tr>
<td>( N_b )</td>
<td>number of bits per term</td>
<td>50</td>
</tr>
<tr>
<td>( N_{fw} )</td>
<td>number of words per fragment</td>
<td></td>
</tr>
<tr>
<td>( N_s )</td>
<td>number of blocks</td>
<td></td>
</tr>
<tr>
<td>( N_t )</td>
<td>number of words of title and author information</td>
<td>50</td>
</tr>
<tr>
<td>( N_w )</td>
<td>number of words per document</td>
<td>10,000</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>( p(w) ) or ( p )</td>
<td>probability that word ( w ) occurs</td>
<td></td>
</tr>
<tr>
<td>( p_n(w) ) or ( p_n )</td>
<td>probability that word ( w ) occurs in fragment or document of length ( n )</td>
<td></td>
</tr>
<tr>
<td>( p_n(w_1, \ldots, w_m) )</td>
<td>probability that fragment or document of length ( n ) contains all of the words ( w_1, \ldots, w_m )</td>
<td></td>
</tr>
<tr>
<td>( pois_n )</td>
<td>Poisson approximation for ( p_n )</td>
<td></td>
</tr>
<tr>
<td>( P(X = v) )</td>
<td>probability Poisson variable ( X ) equals ( v )</td>
<td></td>
</tr>
<tr>
<td>( R )</td>
<td>number of tuples in a relation</td>
<td></td>
</tr>
<tr>
<td>( S )</td>
<td>number of bits in signature per term 32</td>
<td></td>
</tr>
<tr>
<td>( S_f )</td>
<td>size of fragment tuple in bits 5,064 (( F = 100 ))</td>
<td></td>
</tr>
<tr>
<td>( S_k )</td>
<td>size of foreign keys in bits 32</td>
<td></td>
</tr>
<tr>
<td>( S_m )</td>
<td>size of monolithic tuple in bits 505,000</td>
<td></td>
</tr>
<tr>
<td>( S_s )</td>
<td>size of segmented tuple in bits 8,000 (( F = 100 ))</td>
<td></td>
</tr>
<tr>
<td>( S_t )</td>
<td>size of title tuple in bits 8,200 (( F = 100 ))</td>
<td></td>
</tr>
<tr>
<td>( T_d )</td>
<td>seconds to process one bit of data ( 10^{-6} )</td>
<td></td>
</tr>
<tr>
<td>( T_i )</td>
<td>seconds to process one bit of index ( 10^{-7} )</td>
<td></td>
</tr>
<tr>
<td>( T_s )</td>
<td>seconds per seek+latency ( 10^{-2} )</td>
<td></td>
</tr>
<tr>
<td>( v )</td>
<td>number of Poisson trials with particular outcome</td>
<td></td>
</tr>
<tr>
<td>( V )</td>
<td>number of words in a collection that have approx same probability ( p )</td>
<td></td>
</tr>
<tr>
<td>( W )</td>
<td>number of distinct words in database 500,000</td>
<td></td>
</tr>
<tr>
<td>( W(n) )</td>
<td>number of distinct words in document or fragment of ( n ) words</td>
<td></td>
</tr>
<tr>
<td>( x )</td>
<td>independent variable for regression</td>
<td></td>
</tr>
<tr>
<td>( x_1, \ldots, x_l )</td>
<td>series of ( x ) values</td>
<td></td>
</tr>
<tr>
<td>( y )</td>
<td>dependent variable for regression</td>
<td></td>
</tr>
<tr>
<td>( y_1, \ldots, y_l )</td>
<td>series of ( y ) values</td>
<td></td>
</tr>
</tbody>
</table>
Index

atom, 47
atomic attribute, 54
atomic schema, 44
attribute, 26
bit slicing, 158
bit string, 157
block descriptor, 158
Boolean queries, 10
Boyce-Codd normal form, 40
composite attribute, 28
composite schema, 45
conceptual retrieval paradigm, 11
data model, 22
data object, 45
database schema, 23
database state, 23
descriptive markup, 5
document, 2
document collection, 3
document database, 3
document database system, 4
document fragment, 2
domain-key normal form, 41
duplex schema, 162
entity, 26/entity key, 28
exploration paradigm, 10
external level/schema, 95
functional dependency, 38
Horn clause, 47
hypermedia system, 13
hypertext database system, 12
integrity constraints, 24
join dependency, 40
key, 38
link, 11
literal, 47
logical data model, 22
logical level/schema, 95
mapping schema equivalence, 97
matching paradigm, 8
monolithic schema, 162
multi-organizational scheme, 159
multimedia document, 13
multiple-object-reference attribute, 56
multivalued attribute, 28
node, 11
object class, 54
object-reference attribute, 54
parent-child relationship type, 33
partitioned normal form, 44
physical data model, 22
physical level/schema, 95
presentational markup, 5
procedural markup, 5
project-join normal form, 40
punctuational markup, 5
query by document content, 171
query by fragment content, 171
query by title, 169
query descriptor, 157
record descriptor, 157
record type, 33, 36
relation, 38
relationship, 29
schema with bidirectional references, 115
schema with simple acyclic references, 115
schema without references, 115
segmented schema, 162
sequence, 44
set type, 37
similarity matching, 10
simpler query, 107
template dependency, 40
term, 47
term signature, 156
text document, 3
text fragment, 3
tuple, 44
virtual record type, 34