Extending Functional Databases for use in Text-intensive Applications

by

Simon N Sheldrake

Doctoral thesis submitted in partial fulfilment

of the requirements for the award of

Doctor of Philosophy

of Loughborough University

August 2002

Abstract

This thesis continues research exploring the benefits of using functional databases based around the functional data model for advanced database applications—particularly those supporting investigative systems. This is a growing generic application domain covering areas such as criminal and military intelligence, which are characterised by significant data complexity, large data sets and the need for high performance, interactive use. An experimental functional database language was developed to provide the requisite semantic richness. However, heavy use in a practical context has shown that language extensions and implementation improvements are required—especially in the crucial areas of string matching and graph traversal. In addition, an implementation on multiprocessor, parallel architectures is essential to meet the performance needs arising from existing and projected database sizes in the chosen application area.

The work described deals with the general topic of devolving functionality to a lower level in the query evaluation process. It builds on earlier work to show that substantial performance gains are possible in many areas. It then pays particular attention to string handling and the data structures supporting this to provide a richer set of search options for the user—options hitherto unavailable in a functional database. By exploiting the inherent parallelism in list comprehensions—and the optimisations that are available—it is possible to provide language extensions based around a parallel architecture. This architecture uses the basic principles of dataflow graphs, loosely coupled MIMD machines, together with a novel RAID configuration combining mirroring and parity schemes, and obviates the need to maintain complex, low-level indexes. Attribute data is stored separately from entity data, combining the benefits of the functional data model with those of the relational data model to reflect more naturally data usage and provide a further boost to graph traversal operations.

Initial results are promising and show that combining mature technology with novel ideas is achievable without compromising the known advantages of the functional paradigm. The results of this, and other, research work should provide added impetus in making functional databases a more realistic choice for use in advanced, text-intensive database applications.

Keywords: functional data model; functional database; functional programming; function graph model; RAID; parallel processing; dataflow; text searching; triple store

Acknowledgements

The following people have helped and encouraged me in different aspects of my work and I offer sincere thanks to them all. At Birkbeck College, University of London, Professor Peter King, Fefie Dotsika and Jonathan Lawder. At Loughborough University, technical staff Jeff Pitchers, Gurbinder Samra and André Schappo. For administrative support thanks to Irene and Debby, and for emotional support and putting up with me in our shared office for the last two years, special thanks to Marina. For academic help in the Department of Computer Science during various stages of my work, thanks are due to Colin Machin, John Cooke, Roger Stone and my Director of Research Chris Hinde. Special thanks go to my Supervisor, Professor Victor Maller, for all his support, encouragement, patience and kindness, and for arranging the necessary finance through ICL and EPSRC. This thesis is dedicated to my late and dear friend Katie. Without her, none of this would have been possible.

Table of contents

Abstract

Acknowledgements i

Table of contents ii

List of figures viii

List of tables x

Chapter 1 Introduction to the thesis 1

1.1 Introduction 1
1.2 Context of the thesis 2
1.3 Background to the thesis 4
1.4 Thesis methodology 5
1.5 Contribution of the thesis 5
1.6 Limitations of the thesis 6
1.7 Thesis structure 6

Chapter 2 The Current Implementation 9

2.1 Introduction 9
2.2 The software triple store 9
2.2.1 The storage of triples 10
2.2.2 The parameter driven splitting policy 11
2.2.3 Parameter driven merging and reorganisation policies 12
2.2.4 The inverted directory array 13
2.3 The lexical token converter 16
2.4 Interface functions 20
2.4.1 File utility operators 21
2.4.2 Update only operators 21
2.4.3 Query only operators 22
2.4.4 Operators provided by the LTC software 22
2.5 File management 23
2.6 Discussion 26
2.6.1 String matching 26
2.6.2 Tokenisation 27
2.6.3 Functionality 28
2.6.4 The data model 29
2.6.5 Directory structure 31
2.7 Summary 33

Chapter 3 Background to areas of work covered 34

3.1 Introduction 34

3.2 A review of similar alternatives 34

3.2.1 Grid file variants 34
3.2.2 Other binary relational storage structures 37
3.2.3 A triple store based on space-filling curves 39
3.2.4 Conclusions for grid file and BRSS applicability 42

3.3 Text searching and user requirements 43

3.3.1 An introduction to text searching 43
3.3.2 What users expect 45
3.3.3 Categories of users 47

3.4 A review of storage and access methods 48

3.4.1 File structure design issues 48
3.4.2 Main memory databases or disk-based databases? 51
3.4.3 Summary of design considerations 52

3.5 Using a search engine 53

3.5.1 Record structure 55

3.6 Parallel processing 56

3.6.1 Implementing problems in parallel 56
3.6.2 The choices for parallel implementation 59
3.6.3 The vigorous parallelism of AGNA 65

3.7 Introducing redundancy 66

3.8 The physical model for data storage 67

3.9 Introduction to our proposals 70

3.9.1 The data model 70
3.9.2 Improving string handling 70
3.9.3 Extending interface functionality 71
3.9.4 Making the architecture parallel 71

3.10 Summary 71

Chapter 4 Enhancing string manipulation 72

4.1 Introduction 72

4.2 Enhancements to the current software 72

4.2.1 Comparisons between object-level and user-defined functions .. 80
4.2.2 Comparisons between conjunctive and disjunctive search types 81
4.2.3 String searching techniques used 83

4.3 A different approach to string handling 84

4.3.1 A new type for large text documents 85
4.3.2 Combining strings and tokens 86

4.4 The data structures 92

4.4.1 The string tables 92
4.4.2 The string triples 93

4.4.3 Other lexical types 94
4.4.4 The attribute records 95
4.4.5 The structure of text documents 95
4.4.6 Recent developments in text structuring 99
4.4.7 Pre-processing of documents 100
4.5 Description and results 100
4.5.1 Basic structure 100
4.5.2 Description of search algorithms 102
4.5.3 Results 104
4.6 Discussion 107
4.6.1 Functional data languages 108
4.6.2 Object-oriented database systems 109
4.6.3 Convergence of the functional, object-oriented and relational approaches to database systems 110
4.7 Summary 112
4.7.1 Devolving string matching functionality 112
4.7.2 Handling strings differently 113

Chapter 5 Extending interface functionality 116
5.1 Introduction 116
5.2 Extending the language level functionality 117
5.3 Extending the operator level functionality 118
5.4 Extending the storage level functionality 119
5.4.1 Introduction 119
5.4.2 Update operators 120
5.4.3 Schema data 120
5.4.4 Instance data 127
5.4.5 Extensional primary functions 127
5.4.6 Intensional primary functions 129
5.4.7 Secondary functions 130
5.4.8 Retrieval operators 131
5.4.9 File utility operators 133
5.5 Discussion 133
5.6 Summary 135

Chapter 6 An architecture to support parallel processing 136
6.1 Introduction 136
6.2 Using a proven parallel processing architecture design 136
6.2.1 The NCR/Teradata DBC/1012 database machine 137
6.2.2 The NCR 3700 database machine 140
6.2.3 Adopting this architecture to suit our needs 140
6.3 A new RAID level 144

6.3.1 Our RAID approach 145
6.3.2 Improved search times 148
6.4 Combining records and triples 150
6.4.1 Entity triples 151
6.4.2 Attribute records 151
6.5 Our architecture 153
6.5.1 Distribution of data 154
6.5.2 Computation 155
6.5.3 The data processing elements (DPE) 158
6.5.4 The control processing element (CPE) 161
6.6 Search strategies 163
6.6.1 Membership triples 163
6.6.2 Other meta data 163
6.6.3 String tables 163
6.6.4 String triples 164
6.6.5 Triples and records 165
6.7 Discussion 165
6.7.1 The choice of a MIMD architecture 166
6.7.2 Why RAID? 167
6.7.3 The storage model 167
6.7.4 The ADMS model 168
6.7.5 The associative model of data 170
6.7.6 The Universal data model 173
6.8 Summary 173

Chapter 7 Transformation, optimisation and translation 174

7.1 Introduction 174
7.2 Transformation to the sub-language 175
7.3 Optimisations 177
7.3.1 Passing down filters and generators 179
7.3.2 Inverse functions 180
7.3.3 Selection functions 182
7.3.4 High hit rates for lexical values 183
7.3.5 Text searching functions 184
7.4 The abstract reduction machine 185
7.4.1 The reduction process 186
7.4.2 A reduction example 189
7.5 Translation into dataflow graphs 193
Optimisations to DFGs 198
7.5.2 Translation to parallel code 202
7.5.3 Graph analysis 211
7.5.4 Allocation of frame slots 214
7.5.5 Mapping DFGs to parallel code 215

7.6 Performance improvements 218
7.7 Discussion 220
7.7.1 Using a core sub-language 220
7.7.2 Optimisations 221
7.7.3 Why use dataflow? 221
7.7.4 Comparison with AGNA 222
7.7.5 Parallel Haskell 223
7.8 Summary 223

Chapter 8 Database creation, population and maintenance 225
8.1 Introduction 225
8.2 Creating a new database 225
8.3 Populating a new database 227
8.3.1 Inserting membership triples 227
8.3.2 Inserting instance data 229
8.4 Attribute records 230
8.4.1 Id_field 232
8.4.2 Entity field 232
8.4.3 Relations field 232
8.4.4 Attributes 232
8.4.5 Timestamp 233
8.4.6 Text type 233
8.4.7 BLOB type 234
8.4.8 Multi-valued attributes 234
8.4.9 Default values and intensional values 235
8.5 Storage requirements and placement on disk 235
8.5.1 Attributes 235
8.5.2 String tables and lexical triples 239
8.5.3 Text attributes 240
8.5.4 Entity triples 241
8.5.5 Documents 243
8.5.6 Total storage requirements 244
8.6 Indexing 246
8.6.1 Schema triples 246
8.6.2 Membership triples 247
8.6.3 Entity triples 247
8.6.4 Attribute records and documents 247
8.6.5 String tables 248
8.6.6 Lexical triples 248
8.6.7 Temporal indexing 249
8.7 Adding and deleting data 253
8.7.1 Membership triples 254
8.7.2 Attribute records and documents 255
8.7.3 Entity triples 255

8.7.4 Lexical triples and string tables 255
8.7.5 Schema data 256
8.8 Other DBMS related issues 257
8.8.1 Data integrity 257
8.8.2 Concurrent access 258
8.8.3 Security 259
8.9 Summary 259

Chapter 9 Summary, conclusions and further work 261
9.1 Introduction 261
9.2 Statement of the problem 261
9.3 The solution 263
9.4 Proof of solution 266
9.5 Conclusions 270
9.6 Further work 271
9.6.1 Build complete system 271
9.6.2 Partially structured data 272
9.6.3 Optimisation, transformation and searches 273
9.6.4 Hybrid RAID systems 274

References 276
Appendix 290
A.1 North Yorks crime database – schema 290
A.2 North Yorks crime database – triple allocation 291
A.3 Presented at IDEAS '98 Conference, Cardiff UK, July 1998 292
A.4 Presented at BNCOD, Exeter UK, July 2000 294
A.5 'C' program files for text searching based on using records 298
A.6 'C' program files for text searching using 5.5 million records 308
A.7 Initial functions used for experimental searches 317
A.8 Final search functions used in software 329

List of figures

Figure 1.1. The TriStarp Architecture. 3
Figure 2.1. Mapping from logical data space regions to physical disk storage. 11
Figure 2.2. Merging schemes. 13
Figure 2.3. The grid array. 14
Figure 2.4. IDA data structures 15
Figure 2.5. The R-module data structure 16
Figure 2.6. Format for storage of Short strings as string triples. 18
Figure 2.7. Short string internal storage. 19
Figure 2.8. The Triple Store architecture. 25
Figure 2.9. Graph traversal for attributes. 30
Figure 3.1. The BANG file scheme. 35
Figure 3.2. The temporal grid file 36
Figure 3.3. The Hilbert Curve 40
Figure 3.4. First and second order curves. 41
Figure 3.5. States the Hilbert curve can take 41
Figure 3.6. Outline of CAFS. 54
Figure 3.7. Typical record format 55
Figure 3.8. MIMD configuration. 57
Figure 3.9. Parallel processing terminology 58
Figure 3.10. The ADMS data model. 69
Figure 4.1. Crime database schema (part of). 79
Figure 4.2. Improvements in string functions. 80
Figure 4.3. Comparisons between different *or* and *and* functions. 82
Figure 4.4. Document mapping. 96
Figure 4.5. Mapping between document and document file card. 98
Figure 4.6. Overall string matching results. 105
Figure 4.7. Search times with and without a data filter. 106
Figure 6.1. The Teradata DBC/1012 architecture 138
Figure 6.2. RAID 100 disk layout. 144
Figure 6.3. RAID 15 configuration. 145
Figure 6.4. Schema for RAID example. 146
Figure 6.5. Outline of storage model. 151
Figure 6.6. Outline of our architecture. 153
Figure 6.7. Structure of an open list. 156
Figure 6.8. Construction of filtered lists. 157

Figure 6.9. Graph traversal across DPEs 159
Figure 6.10. The Control Processing Element. 161
Figure 6.11. Mapping whole string. 164
Figure 6.12. Mapping truncated string. 165
Figure 6.13. The associative model of data. 171
Figure 6.14. Sentences inverse functions. 172
Figure 7.1. Arithmetic DFG. 194
Figure 7.2. Example of a DFG procedure. 196
Figure 7.3. Conditional DFG. 197
Figure 7.4. DFG for *apply*. 198
Figure 7.5. DFG for *built-in* function. 200
Figure 7.6. DFG for string lookups. 201
Figure 7.7. DFG for *restrict*. 202
Figure 7.8. Organisation of the parallel machine. 203
Figure 7.9. Methods of dependence sets. 213
Figure 7.10. Procedure frame 214
Figure 7.11. The *lookup* operator. 216
Figure 7.12. The *string-map* operator. 216
Figure 7.13. The *subtract* operator. 216
Figure 7.14. The *restrict* operator. 217
Figure 8.1. Raw data schema 230
Figure 8.2. North Yorks crime database – Triple allocation (part of). 237
Figure 8.3. Storage architecture. 246
Figure 8.4. Schema changes 256

List of tables

Table 2.1. Allocation of type labels to token space. 17
Table 4.1. Current built-in string functions.. 73
Table 4.2. Experimental string functions. 77
Table 4.3. Object-level string functions. 78
Table 4.4. Comparisons between *contains* and *em1*. 80
Table 4.5. Timings for different types of *or* and *and* functions........................ 82
Table 4.6. String table record structure. 93
Table 4.7. Example of string table.. 93
Table 4.8. String triples. 94
Table 4.9. Search types. 101
Table 4.10. Forward names file.. 101
Table 4.11. Search patterns and timings.. 106
Table 5.1. Record identifiers. 121
Table 5.2. Type identifiers... 121
Table 5.3. Contrast between primary and secondary functions.................... 124
Table 5.4. Schema table. 126
Table 6.1. Schema table for RAID example. 146
Table 6.2. RAID triples.. 147
Table 6.3. Improved search times using RAID 15. 149
Table 6.4. Comparison of popular RAID levels. 149
Table 6.5. Triples used in single processor example. 160
Table 8.1. New type labels... 226
Table 8.2. Example of attribute data values. 231
Table 8.3. Description of record fields... 231
Table 8.4. Attribute record storage requirements (in bytes). 239
Table 8.5. Lexical triple store requirements... 241
Table 8.6. Total document storage requirements... 244
Table 8.7. Total storage requirements. 245
Table 8.8. Storage requirements for temporal index. 251
Table 8.9. Optimised temporal index requirements...................................... 252
Table 8.10. Triples and records for schema changes. 256

Chapter 1 Introduction to the thesis

1.1 Introduction

Functional programming languages have been in use for over twenty-five years fulfilling many early promises made by their supporters. Indeed, in the 1980s, the functional approach was seen as the answer to numerous problems in computing and gave rise to considerable research in many countries throughout the world. In particular, their appeal as elegant ways to provide parallel processing generated great interest. However, some of the initial hopes had to be modified because of discovered drawbacks. Specifically, handling input/output and updates have remained difficulties. Furthermore, the task of implementing a parallel functional language is much more substantial than it first appears [TRI96].

Where database systems are concerned, the functional option is just one of a number of paradigms available to developers and is not usually the first choice. Increasingly, commercial database systems need to handle large volumes of text efficiently, and text handling has never been a particular strength of functional languages. However, as the shortcomings of relational databases become increasingly noticeable, and object-oriented systems fail to make a significant impact commercially [WIL00], there is still room for the functional approach. More optimistically, a functional basis can be used to provide a stricter formalism to underpin a synthesis of the relational and object models—thus combining the best of both worlds. This is beginning to happen and is evident in the recent standard for SQL:1999 [MEL02].

This thesis aims to add impetus to this argument proving that enhancements in certain areas are achievable without compromising the earlier benefits. The work described in this thesis forms part of a collaboration with the TriStarp project, to which an introduction and background is given next.

1.2 Context of the thesis

The TriStarp—Triple Store Applications Research Project—started under the guidance of Professor King at Birkbeck College, University of London in 1984 and has used the binary relational approach for its database development since then [KIN90]. Starting from the basic storage level and moving up to the user interface and conceptual views, full functional database systems can be developed which include deductive facilities, integrity constraints, temporal data, and provision for modelling and storing arbitrarily complex objects. Also database systems can have a higher level of data independence than has hitherto been achieved. Increases in hardware power and improved software techniques have the potential to make such systems practically realisable for the next generation of database management systems.

Frost reviewed the binary relational approach and devised the Binary Relational Storage Structure (BRSS) [FRO82]. A BRSS holds data in three fields with the format `<subject, relation, object>` termed a triple. A triple can hold facts like "Fred reads The-Times", or be used for structures like binary trees—where format `<node, left-subtree, right-subtree>` is used to construct a triple set. There were several developments of the BRSS concept and the Birkbeck Triple Machine (BTM) is one such implementation [DER89]. This is described in chapter 2.

The functional database language FDL [POU92] was the culmination of several years' research work into functional programming done at various British universities. It remedied earlier drawbacks by unifying the functional data model with functional programming. This allows it to be used for effective modelling as well as computation. From a database viewpoint, FDL has as its model the functional view of the binary relation model usually known as the functional data model. In a binary relational model, entities may be lexical—they can be written down, viewed or printed—such as string or integer; or they may be non-lexical. Non-lexical entities cannot be viewed or expressed directly, they may only be referred to by the (lexical) attributes that define them.

The advantage of this model is that people without any prior mathematical knowledge or special training may easily understand it. As a consequence, it is a persuasive data model for capturing real-world semantics of a domain as well as being able to represent schema information in a simple diagrammatic form for easy comprehension.

Issues tackled earlier in the project include temporal dependency, the ability to handle intensional as well as extensional function definitions and the ability to handle incomplete or unknown information. The use of a functional data model allows for extensible schemata, while the benefits of functional programming include the ability to support constructed types and recursive functions over them. The TriStarp architecture is shown in Figure 1.1 below.

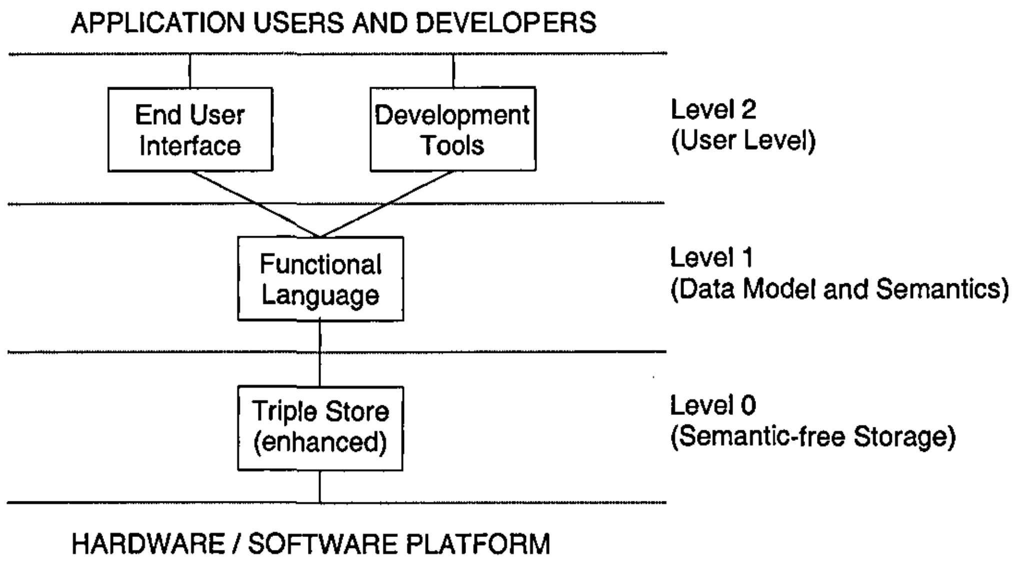

APPLICATION USERS AND DEVELOPERS
End User Interface
Development Tools
Level 2 (User Level)
Functional Language
Level 1 (Data Model and Semantics)
Triple Store (enhanced)
Level 0 (Semantic-free Storage)
HARDWARE / SOFTWARE PLATFORM

Figure 1.1. The TriStarp Architecture.

1.3 Background to the thesis

Previous investigations by Professors Maller and King evaluated the suitability of a functional database language being used to support large applications in the field of investigative systems [KIN96a]. This is a growing generic application area covering criminal and military intelligence and characterised by significant data complexity, large data sets, and the need for high performance, interactive use [MAL96].

The evaluation confirmed the soundness of this approach but heavy use in a practical context showed that language extensions were needed, together with implementation improvements, particularly in the areas of string manipulation and graph traversal. Also, an implementation on multiprocessor, parallel architectures was considered necessary to meet the performance requirements arising from existing and projected database sizes in this application area.

Five objectives from the above investigation were achieved but completion of the following areas remain outstanding:

(a) refine, extend and re-specify the abstract machine interfaces
(b) reassess implementation methods, particularly the use of parallel architectures to boost performance at level 0
(c) further investigate, refine and re-specify level 2 facilities.

The subject of this thesis is to investigate improvements in areas (a) and (b) above. The three main areas of investigation are string handling and improved searching facilities, graph traversal and the incorporation of parallel processing techniques and enhancing the abstract machine level interface (and general) functionality. The initial investigation did, however, lead to other areas of work that are detailed below.

1.4 Thesis methodology

This study combines findings from different, but inter-related, disciplines of computer science. These are:

- functional programming
- data models
- redundancy
- parallel processing and dataflow
- text processing and data filtration, and
- database management systems.

The literature review is therefore presented in two different forms. The bulk of this is contained in the second and third chapters of the thesis. Then later chapters extend upon this where necessary to set the scene for each topic that the chapter covers. Each of the chapters 4 to 8 autonomously tackles an area of the work giving individual results or examples as proof of concept where appropriate. The last chapter draws together the various components of the thesis into appropriate conclusions.

1.5 Contribution of the thesis

More than one area of work is described in this thesis. As set out in the methodology above, there are several inter-related disciplines involved and contributions are made to the following areas.

Architecture – The combination of attribute records and entity triples for physical storage seems to complement naturally the identical graphical representation that the user sees and uses to form queries. This approach also aims to synthesise the benefits of the relational data model with those of the functional data model.

String manipulation – The removal of strings from the data model and associated physical storage constraints would seem a sensible thing to do. This allows for the provision of more powerful string handling operations and the inclusion of a potentially richer type system, e.g. for type *text*. This permits further diversity of functionality to differentiate between string and text operations.

Redundancy – Using a combination of mirroring and parity—where the mirror complements inverse functions that are part of the data model—a novel RAID level more naturally supports the new architecture of attribute records and entity triples. This is in addition to a much greater level of data security provided.

Functionality – Devolving functionality is taken a stage further than has hitherto been done in functional languages—specifically in relation to string manipulation and inverse function evaluation.

1.6 Limitations of the thesis

Although they clearly complement each other, the contributions detailed above have been achieved largely in isolation. There is no all-inclusive working system that incorporates all of these concepts. This would have proved difficult to achieve in this instance as the areas of contribution cover a wide range of disciplines. Moreover, the work involved in writing a complete working system would not have contributed directly to the areas of work themselves—each of which is considered and evaluated in its own right.

1.7 Thesis structure

The first three chapters constitute the introduction and background to the work presented in the later chapters of the thesis. A synopsis of each chapter is given below.

Chapter 1 (this chapter) is the introduction to the thesis.

Chapter 2 introduces the current triple store architecture including the lexical token converter and interface functionality before identifying where shortcomings are evident and where improvements could be made.

Chapter 3 begins by discussing alternative implementations based on similar architectures. User requirements are then considered followed by a review of storage and access methods. Parallel processing techniques are an important area of the work presented later on, so the basic concepts are introduced here. The chapter concludes with an introduction to the topics that form the remaining chapters of the thesis.

Chapter 4 specifically tackles one of the main areas of investigation, namely, improvements to string manipulation. This is done in two stages. The first stage involves providing improvements within the confines of the current architecture giving results and making comparisons where appropriate. The second stage suggests an alternative approach using different data structures to provide improved searching opportunities. This strategy is then compared to other systems.

Chapter 5 investigates areas for improving functionality with specific emphasis on the interface functions between levels 0 and 1 of the software hierarchy shown in Figure 1.1.

Chapter 6 covers another main area of investigation in this work, namely, architecture. A brief introduction is given to the NCR/Teradata database machine configuration that serves as a model for the proposed architecture. A novel RAID level is then described by example and compared to other RAID levels. The concept of combining records and triples is followed through again with examples before the choices made are put into context of other work.

Chapter 7 goes into the detail of transformations, optimisations and translations that are made to user expressions. The abstract reduction machine is described and the translation into dataflow graphs is shown by example. An important area of this work is the devolution of functionality to a lower level in the evaluation process. This is described in the context of previous research and, in particular, to the string and text enhancements (and inverse functions) introduced earlier in this thesis.

Chapter 8 uses figures extrapolated from the North Yorks Crime Database and covers creation, population and maintenance issues. An important aspect missing from the original TriStarp proposals was temporal indexing, so this is described here. Finally, other database issues are summarised.

Chapter 9 gives a summary and conclusions including related work and further work arising from the thesis.

Throughout this thesis the words *entity, non-lexical, abstract entity* and *object* will be used synonymously. The terms *lexical, lexeme* and *attribute* will be used likewise, as will the terms *relation* and *function.*

Chapter 2 The Current Implementation

2.1 Introduction

In this chapter we begin by describing the storage architecture used in the TriStarp software—with particular emphasis on the storage of triples and the directory structure with its optimisations employed. We then briefly detail the mapping used for creating the lexical and non-lexical tokens used in the triple store together with the interface functions and file management system used. Finally, we discuss the current implementation highlighting areas where improvements could be made.

2.2 The software triple store

The triple store is essentially an abstract machine that provides two facilities: a method of storing triples and a set of interface functions to manipulate the triples. The triple store is based on the concept of a Binary Relational Storage Structure (BRSS) [FRO82]. All data are held as triples with the format `<subject, relation, object>` and the valid operations are: *insert_a_triple, retrieve_set_of_triples* and *delete_set_of_triples*. For simplifying queries the notation '*' for a known value and '?' for an unknown value is used. Thus there are seven Simple Associative Forms (SAFs) available to manipulate the triples:

```
<*,?,?>  <?,*,?>  <?,?,*>  <*,*,?>  <*,?,*>  <?,*,*>  <*,*,*>
```

(`<?,?,?>` is an eighth but serves no purpose as it means dump the whole store.)

The advantages of a BRSS include simplification of design and use and the improvement of data independence [MCK92]: a disadvantage is the need to hold more triples to store the same information than would be held in a record-based, relational system. One implementation of a BRSS is the Birkbeck Triple Machine (BTM). BTM comprises two components: a software Triple Store and a Lexical

Token Converter (LTC). The function of the triple store is the storage, retrieval and deletion of triples (note that the BTM holds triples in the form `<relation, subject, object>`). The LTC handles the mapping of the external representation of the components of a triple (strings, integers, etc.) into an internal representation of fixed-length identifiers (32 bits for each identifier). Internal representation of a triple therefore appears as 96 bits. The LTC also handles the reverse conversion when triples are requested by users. A set of functions is provided that enables the use of a uniform interface for triple store access. Indeed, as the BTM is semantic-free, it has been used effectively in functional and logical database projects at Birkbeck.

2.2.1 The storage of triples

The data structure behind the BTM is based on the grid file [NIE84]—a dynamic, multi-dimensional method of data organisation within a file. The logical data space available for the storage of triples is determined by taking the Cartesian product of the domain of each of the three key attributes—thus the BTM is a three-dimensional grid file.

When a new triple store repository is opened, the whole of the data space region represents (points to) a data page on disk. As the file expands—i.e. triples are inserted into the file—the data space will require division into new regions as data pages fill up and new ones are required. The regions are divided into hyper-rectangular sub-spaces by repeated bisection of the domains of attribute values in one of the three dimensions. Similarly, when triples are deleted from the file, merging of two pages into one may be possible and thus sub-space regions can be combined along the same lines. There is a direct mapping between data space regions and data pages on disk, which is shown in Figure 2.1.

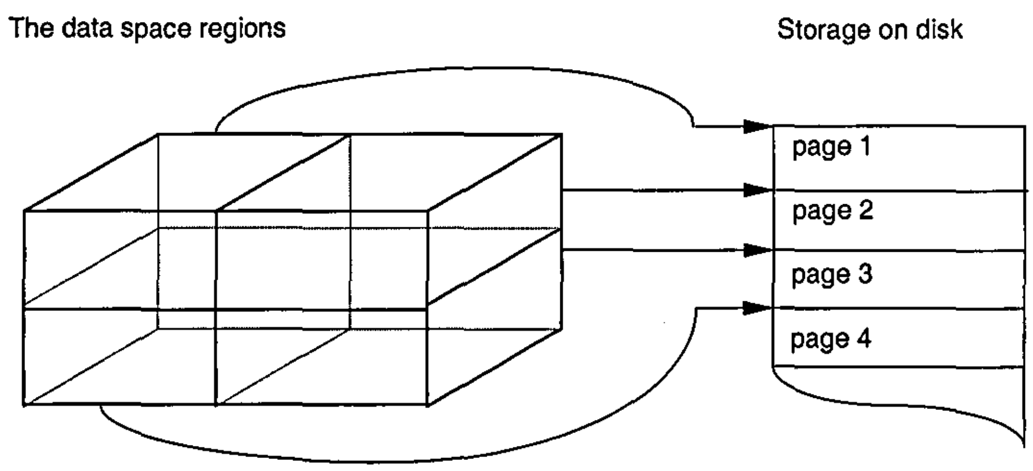


Figure 2.1. Mapping from logical data space regions to physical disk storage.

There is a threshold of page occupancy—optimally set at around 70%—which, when reached, requires that the page be split into two new pages. Similarly, when page occupancy falls below a threshold—optimally 30%—a check is made to see if any merging of regions (and thus data pages) can take place. So maintaining an effective, dynamic triple store depends on efficient splitting and merging operations performed on data space regions to reduce data page accesses—which are heavily I/O-bound—to a minimum. There is also the problem of trying to map triples from multi-dimensional data space, on to a one-dimensional storage medium so that triples close together logically are also close together physically. Maintaining a dynamic triple store also depends on the form and granularity of the directory used to access the data pages.

2.2.2 The parameter driven splitting policy

The splitting and merging of regions can be done in several ways. The original method suggested by Nievergelt uses a halving of domain partitions in alternate dimensions. This makes merging easier but takes no account of the highly likely situation of triples clustering in the data space, and can lead to under-populated or empty regions and a complex directory structure. The BTM overcomes this by testing potential split points in a region using a *select_sp_point* algorithm. This chooses the point of split in a region so that the resulting new regions created are much less likely to be empty. This also reduces the size of the directory.

However, the dimension to be split must first be chosen. BTM uses occurrence and reference probabilities known about the data to calculate a cost of split in each dimension; it then effects a split in the dimension of lowest cost [Dera90]. Occurrence probabilities, as to the expected form a partially specified query will take, can be provided by the Database Administrator (DBA). These probabilities are given at the time the triple store is opened and correspond to six of the SAFs described above and are denoted as $Q_{\{1\}}$ for <*, ?, ?> through to $Q_{\{2,3\}}$ for <?, *, *>. Reference probabilities use information on the probability distribution of values specified in queries and are a weaker element of the policy as they have to be hard coded by the DBA so are less easily changed.

The cost of a split is independent of the state of the file and only makes use of the parameters provided by the DBA, although different occurrence probabilities can be given each time the store is opened. Since most partially specified queries have the relation field known, the probabilities provided by the DBA reflect this and so splits are often biased towards partitioning in the relational dimension.

2.2.3 Parameter driven merging and reorganisation policies

The merging of regions also uses the above probabilities provided by the DBA. There are two merging schemes: buddy and neighbour. In the buddy system only one candidate in each dimension can be considered for merging. This candidate is the one that, if merged with, would form a region that could have been obtained by repeated bisections of the domain. BTM uses the less restrictive neighbour system. In this system a region can merge with either of its neighbours in each dimension. (See Figure 2.2 below in two dimensions for clarity.)

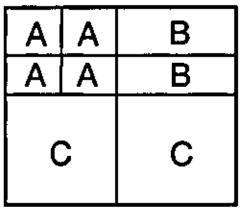


Figure 2.2. Merging schemes.

Buddy system: each region with label *n* is a candidate for merging with a region of the same label (diagonal A's excepted). **Neighbour System**: any region could merge with another so long as the resulting new region is hyper-rectangular. So lower-right A could merge with upper-right A, lower-left A or lower B but not with left C because this would give a non-rectangular region.

The situation can arise when an under-populated region has no legitimate region to merge with because of the configuration of the data space. In this case a low utilisation of the storage space would result—to the obvious detriment of the system performance. This situation is referred to as deadlock [HIN85] and can only be resolved by a reorganisation of some or all of the data space regions. BTM has a parameter driven reorganisation policy to handle deadlock where neighbouring regions in all dimensions are looked at to see if splits in other dimensions would allow for more efficient merges to take place.

2.2.4 The inverted directory array

The maintenance of a directory is crucial to minimising page accesses to secondary storage and has been the subject of much research since the grid file was first proposed in 1984 (discussed further in chapter 3). The grid file directory uses one linear scale for each of the three dimensions to maintain a record of partition points used in the organisation of the file. The Cartesian product of the intervals used forms a three-dimensional array termed the grid array, each element of which is termed a grid block. The grid array is usually too large to be held in main memory—unlike the linear scales used for its partitioning.

Each grid block contains a pointer that either points to a data page in memory, or is a null pointer if the region to which that grid block refers contains no records. The original grid array had many null pointers because of the nature of the cyclic, halving policy used for splitting: at its worst, the grid array could grow exponentially [FRE87]. This is because, as each new data partition is created, a hyper-plane is inserted through the data space creating many new under-populated or empty grid blocks to be added to the grid array. As the BTM uses a splitting algorithm to ensure that empty regions are kept to an absolute minimum, some of the problems of null pointers are removed. However, this still leaves the problem of having too many grid blocks pointing to the same data page. In the two-dimensional Figure 2.3 below, the p# in each grid block points to a data page in memory. There are thus nine data space regions and 16 grid blocks in this example, but only eight data pages in memory.

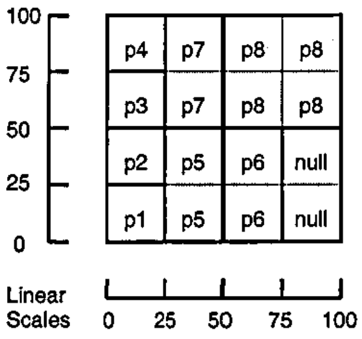


Figure 2.3. The grid array.

As there can be several grid blocks pointing to one data page, BTM uses an Inverted Directory Array (IDA) [DER89] to remove the duplicate pointers. This is done by maintaining a set of pointers for each interval plane. The intersection of two or three of these sets provides a set of pointers that determine the relevant data pages to search. The IDA consists of data structures that facilitate the above. Linear scales are referred to as extended scales (E-scales); the set of pointers for each interval of each E-scale is referred to as a set record (S-record). The collection of S-records forms the set module (S-module). (See Figure 2.4 below.)

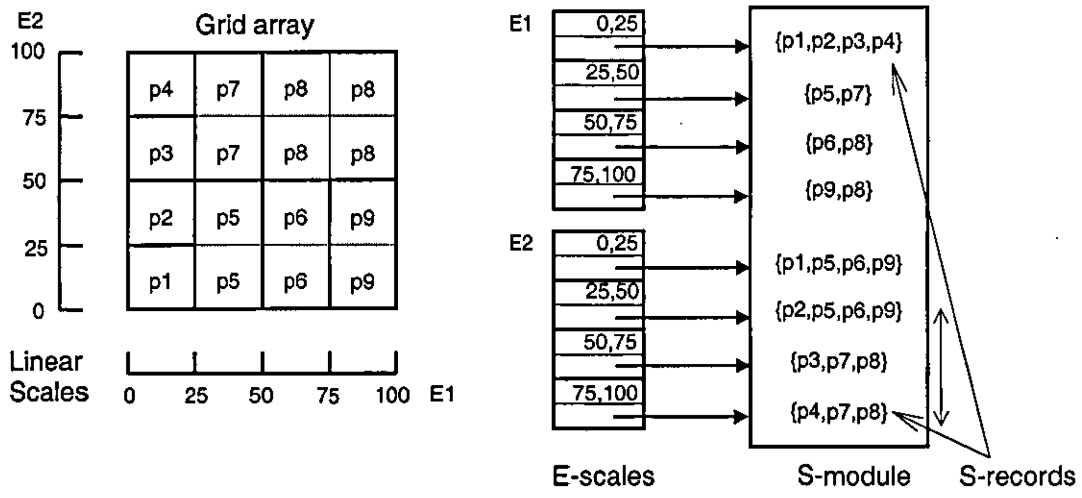


Figure 2.4. IDA data structures.

Following the example above, there are eight duplicate page pointers removed from the directory structure by using IDA.

The IDA can effectively map linear scales onto data page pointers using the above data structures. Merging and splitting of data space regions can be done more efficiently without needing a re-write of the whole directory. However, there are occasions when a mapping is required in the opposite direction, e.g., when a data page becomes over- or under-populated.

In this situation the directory wants to know which regions will need to have their linear scales (E-scales) updated. To cope with this, the IDA has another data structure—the region module (R-module)—to hold a list of all the data pages in the file and their interval boundaries. (See Figure 2.5 below for the R-module relative to the previous example.)

<p1,[0,24],[0,24],c>	<p4,[0,24],[75,99],c>	<p7,[25,49],[50,99],c>
<p2,[0,24],[25,49],c>	<p5,[25,49],[0,49],c>	<p8,[50,99],[50,99],c>
<p3,[0,24],[50,74],c>	<p6,[50,74],[0,49],c>	<p9,[75,99],[0,49],c>

Figure 2.5. The R-module data structure.

The 'c' in the above structure refers to the number of triples in each page; it is done to obviate explicitly counting triples each time the page is considered for merging or re-organisation. The price to be paid for this is that the R-module requires updating each time triples are inserted or deleted from the triple store. The R-module is organised as a B-tree structure with two layers: the root is held in main memory and the nodes are held in R-pages—an R-page being the unit of storage in the triple store.

2.3 The lexical token converter

The Lexical Token Converter (LTC) builds on the original ideas of Lavington and Wang [LAV84], who built a hardware LTC for the Intelligent File Store (IFS) [LAV88]. BTM uses a software LTC where each triple consists of three identifiers each of which is 32 bits long. As the domain of 32-bit integers is insufficient to represent instances of all the data types that are required, a conversion of triples from their external representation to internal identifiers is necessary. The LTC uses a set of internal interface functions to maintain a one-to-one mapping between triples and their internal identifier tokens.

In addition to the standard data types of string, integer and real that are referred to as lexical tokens, the LTC also handles non-lexical tokens and system tokens. Non-lexicals are used for the representation of the abstract entities defined by the user and are described by the lexical tokens they form relations with. The abstract entity "person" cannot be accessed directly—only via the functions that describe it (name, date-of-birth, etc.). Non-lexical tokens are maintained by consecutive identifiers. System types are for individual users to tailor for their own particular needs relative to how their software will use the triple store. FDL uses system tokens for constants, labels used in query trees, built-in function identifiers and semantic error codes.

The 32-bit token space is partitioned into disjoint subspaces to allow for the storage of each of the types required. If the token space is considered in 10000000_{hex} blocks, the types are allocated space in the following ratios that are proportional to the expected number of occurrences of each type. These are identified by the most significant of the 32 bits (MSB) which are used as a type label. (See Table 2.1 below.)

type	ratio	range	type label
Short	2	00000000 to 1FFFFFFF	0 0 0 X
Long	2	20000000 to 3FFFFFFF	0 0 X 1
Res-short	2	40000000 to 5FFFFFFF	0 1 0 X
Res-long	2	60000000 to 7FFFFFFF	0 1 1 X
Non-lex	1	80000000 to 8FFFFFFF	1 0 0 0
System	1	90000000 to 9FFFFFFF	1 0 0 1
Integer	2	A0000000 to BFFFFFFF	1 0 1 X
Real	4	C0000000 to FFFFFFFF	1 1 X X

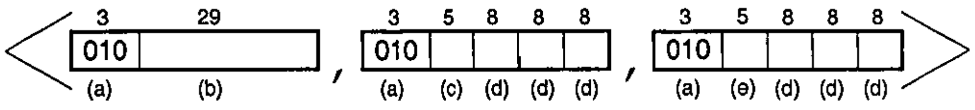
(Where X indicates "don't care".)

Table 2.1. Allocation of type labels to token space.

String types are handled as follows: the first two types, *Short* and *Long*, are used to generate unique, consecutive identifiers. The string can then be broken down into sub-strings and stored as special triples within the subspace identified by the third and fourth types, *Res-short* and *Res-long*. The subspaces 010... and 011... are inaccessible to the user and are referred to here as the reserved subspace for the storage of string triples.

Given a Short string of length *n*, the LTC generates a unique identifier from the domain of type *Short*. A slot system is used to achieve a uniform distribution of string identifiers resulting in a more efficient use of database resources. The identifier space for Short strings is divided into 128 equally spaced slots and identifiers are issued according to a cyclic pattern with the current slot counter incremented as each new identifier is issued. When the store is closed the values of the counters are saved as special triples and re-installed each time the store is re-opened.

Once the LTC has generated a unique identifier, it then proceeds to decompose the Short string into sub-strings for storage in the reserved subspace as string triples. A string of length *n* will require the creation of a set of `n div 6 + 1` string triples, each of which has as its first element the unique identifier (29 bits long) generated by the LTC. In the second and third elements there is room to store, per element, three characters (8 x 3 bits) plus five offset bits together with 010 as the Short string type label. (See Figure 2.6 below.)



(a) Type label
(b) Unique set ID
(c) Offset 1
(d) Single character
(e) Offset 2

Figure 2.6. Format for storage of Short strings as string triples.

The last reserved triple in the chain must have a null byte added to indicate end of string. The concatenation of the offset bits gives the position that the sub-string will take in the complete string. Therefore, the maximum length for a Short string is derived by multiplying the size of the offset by the number of characters held in each string triple less one byte for the null terminator. Thus the maximum length of a Short string is `(2`^{10}`-1) x 6 - 1 = 6,137` characters—about two pages of A4 text with font size 10.

Long strings use a similar scheme to the one described above but with some important differences. Firstly, the third MSB of each string triple (Figure 2.6) is a "1" for all Long strings. Secondly, each triple generated has an additional non-key triple associated with it. The term non-key is used here to describe a triple that is not part of the grid file index system. Hence, these triples are not part of the triple store data space and can take any 32-bit values without corrupting the indexing system. This provides space for an additional 12 characters to be associated with each triple—four characters in each of the three non-key, 32-bit data fields. Thus the maximum length of a Long string is `(2`^{10}`-1) x (6 + 12) - 1 = 18,413` characters—about six pages of A4 text.

A set of internal functions is provided to enable the LTC to perform the storage, retrieval and deletion of string triples that correspond to user manipulation of external triples. Two immediate problems became apparent: there is often much duplication of characters at the beginning of strings and; any string with a length that is a multiple of six characters will need an extra triple in the set that will store six null bytes to terminate the string. Apart from wasting space, this has implications when collecting together a set of strings to search as there will always be a large set of string triples having six null bytes in them. Figure 2.7 shows how the Short strings "simon" and "victor" are stored ("\0" = null byte).

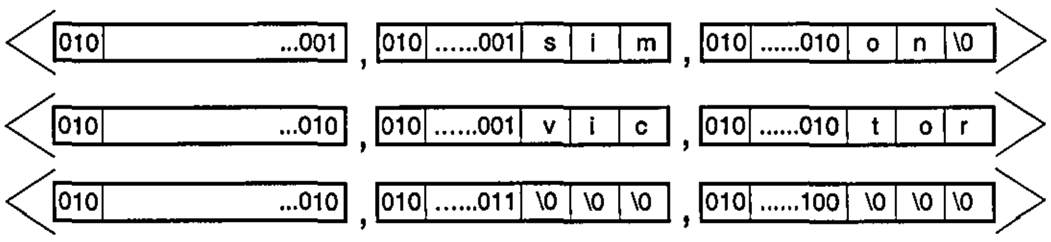

010 ...001 , 010 ......001 s i m , 010 ......010 o n \0
010 ...010 , 010 ......001 v i c , 010 ......010 t o r
010 ...010 , 010 ......011 \0 \0 \0 , 010 ......100 \0 \0 \0

Figure 2.7. Short string internal storage.

The BTM uses two search methods to counteract the above problems: one that searches from the front of a string and one that searches from the end. The criteria used in deciding which method to use is as follows: use reverse search if there is at least two non-null characters in the final reserved triple and the string is composed of more than two reserved triples, otherwise use forward search. A trivial examination of the search string will reveal which method to use.

The storage of integers is handled by straightforward bit manipulation. The three MSBs are needed for the integer label and the fourth MSB is used as a sign bit. Therefore the range of integers available is -2^{28} to $2^{28}-1$.

Reals are stored in a similar way to integers. The two MSBs are needed for the type label and the third MSB for the sign bit. As reals must conform to the IEEE 754 standard for a 32-bit word size, a bit shifting function is used to regain the required length. This results in a small loss of accuracy as two bits of the mantissa are lost during conversion to internal format and padded with zeros when converted back again. One of the problems with the original LTC—maintaining numerical ordering—has been addressed by the BTM implementation.

Non-lexicals are generated from a counter using the domain of the non-lexical type subspace in a similar way to string identifier generation. At the end of each session the counters for non-lexical type and string types are saved as special triples. Non-lexical tokens are completely semantic free: there is no ordering implied regarding the class of entity that they represent.

2.4 Interface functions

To enable the BTM to be used effectively by other software modules, a set of interface functions provides operators for the creation, opening, closing and manipulation of a store. The operators are classified into three categories: file utility, update only, and query only. These categories are now described.

2.4.1 File utility operators

A new triple store can be created with the *ts_create* function that must be supplied with the file name as parameter. Each time the store is opened new values for occurrence probabilities can be provided in the parameters—together with the upper and lower limits for data page occupancy. The form of a call to open a store is:

ts_open(<store_name>,$P_{\{1\}}$,$P_{\{2\}}$,$P_{\{3\}}$,$P_{\{1,2\}}$,$P_{\{1,3\}}$,$P_{\{2,3\}}$,Lim_1,Lim_2)

where each $P_{\{x\}}$ refers to the occurrence probability for each of the six SAFs that a partially specified query $Q_{\{x\}}$ can take—from section 2.2. The sum of the $P_{\{x\}}$ values must of course be equal to one. Lim_1 and Lim_2 are the lower and upper limits for data page occupancy. Finally, *ts_close* will close the store currently in use.

2.4.2 Update only operators

This group provides for the insertion and deletion of triples. The interface functions available are: *ts_insert(triple)* that inserts a triple specified in the only parameter and, *ts_delete(template)* that deletes triples specified in the template provided as the only parameter. A template must match one of the six SAFs and uses the value $-2^{31}+1$ to indicate an unspecified value. An unspecified value can be in one or two of the three fields: the case of all three fields being unspecified is not allowed. There are similar functions for operation on ranges of triples.

2.4.3 Query only operators

The following operators provide facilities for querying various aspects of the triple store that is in operation. In particular, the four operators: *ts_open_set, ts_range_open_set, ts_fetch_another* and *ts_close_set,* provide the means for the retrieval of triples matching a given template.

ts_open_set(template) takes a triple template and returns a unique set identifier that is used in lazy retrieval of the triples. *ts_range_open(template)* operates in a similar way. *ts_close_set(set_id)* merely disassociates the set identifier from the open set. Several retrieval sets can be open at the same time, so the operator *ts_fetch_another(set_id)* will return the next member of the set identified by *set_id*. A further function *ts_present(template)* can be used to retrieve a fully specified triple with format <*, *, *> or, indeed, just to see if a triple matching these fields is present in the store. There are similar operators for the manipulation of ranges of triples where two ranges are provided for each element of a triple—the upper and lower limits of the range to be acted upon.

2.4.4 Operators provided by the LTC software

Very briefly, the operators provided by the LTC software and their purpose are as follows. *ltc_insert_string(str)*—takes a string as parameter and generates a string identifier returning true for success. If a token already exists for the given string then that is returned instead. *ltc_str_to_id(str)*—returns an identifier for the given string. *ltc_id_to_str(id)*—returns a complete string given an identifier as parameter. There are similar 'pairs' of functions for the mapping of integers and reals. *ltc_generate_nonlex*—is guaranteed to return a unique identifier from the domain of non-lexical integers available. The operator *ltc_get_type(id)* will return the type of a given identifier. The LTC software does not currently provide facilities for deleting strings and unwanted non-lexical identifiers.

2.5 File management

The buffer manager is responsible for operations on the buffer pool that comprises a fixed number of slots—each slot having the same size as a page in memory. The number of slots is configurable at initialisation and is not constrained to remain fixed throughout the lifetime of a database. To alter the number of slots the source code must be re-compiled with the new value.

The result of a *fetch_another_page* operation causes the following to happen: a check is first made to see if there are any free slots available in the pool. If there are, then the required page is copied into a free slot. If there are no free slots a page must be dumped from the pool to free a slot for the new page. Pages in the buffer can be FIXED or UNFIXED depending on whether the page is likely to be accessed frequently or not. The buffer manager takes this into account and, whenever a page needs to be dumped from the pool, it first looks for unfixed pages. If none can be found it uses the least recently used (LRU) [KNU73] algorithm to select a page to be dumped to make room for the new page. If the page to be dumped from the buffer pool has been modified since in was copied into the pool, then it is written back to the file before its slot is overwritten by the contents of the new page.

Once a data page has been copied into the buffer pool it can be searched for triples matching a given search template. The triples are ordered within each data page: firstly on the relation field, within that on the subject field and within that on the object field. Because of this, and the fact that the majority of searches specify the relation field or a combination of the relation-subject fields, a binary search technique can be used when either: the first; the first and second; or the first, second and third fields are given. So, using the previously described notation of '*' to indicate a known value and '?' to indicate an unknown value, search templates with the form <*,?,?>, <*,*,?> and <*,*,*> can be matched using a binary chop method of searching.

However, if a search template of <*,?,*> is specified, a linear search would have to be done once the first occurrence of the matching relation field was found. The ordering of triples within a page was a deliberate attempt to improve response time for queries, albeit at the expense of update performance. When a matching triple is found, a pointer to it is returned to the calling function. On the execution of a *fetch_another* instruction, the search continues from the position of the last matching triple. A flag is used to indicate when the search is complete.

Search templates with the form <?,?,*>, <?,*,?> and <?,*,*> cause particular problems in that, because of the ordering of the pages, they are much more likely to produce a large set of page identifiers whose pages may contain a match. When this happens, the page identifiers are themselves held as triples in retrieval pages (see below) and await being passed to the calling function under the implementation of lazy retrieval.

All the information that is kept on disk is stored in one file termed the database file. The database file consists of a header and a set of pages of the same size, with the page size being set on configuration. The header stores the information, such as the E-scales, that resides in virtual storage when the triple store is in operation. At the start of each session, the contents of the header are copied into virtual storage for subsequent use and, at the end of the session, they are written back to the database file. A page in the database file is one of the following five types:

R-page:	holds information required for the region module
S-page:	holds information required for the set module
data-page:	for the storage of individual, unique triples
retrieval-page:	used for storing the identifiers of the data pages that contain possible matches to a query. Such a page is then retrieved lazily
free-page:	not one of the above pages and available for use.

Finally, the TriStarp triple store component is shown in Figure 2.8.

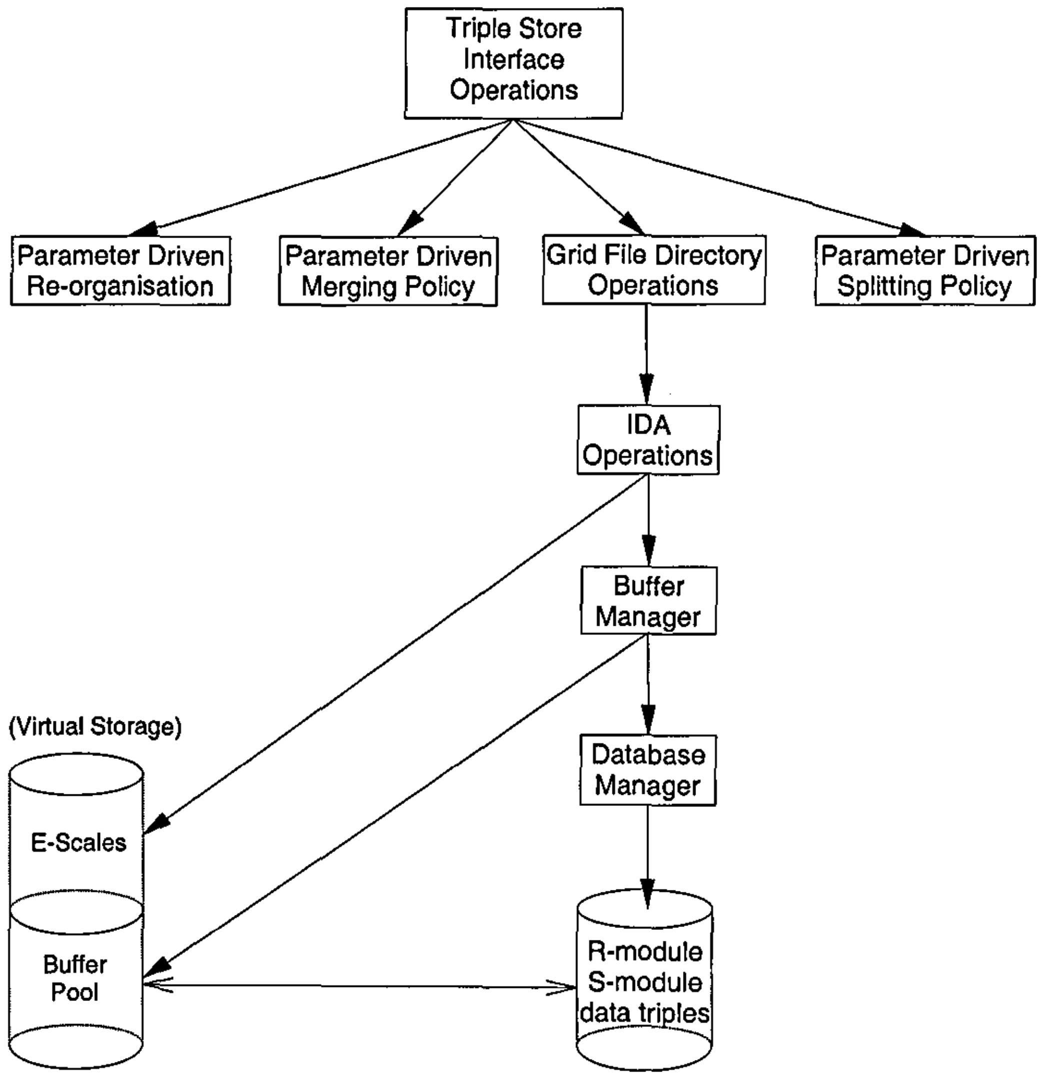

Triple Store Interface Operations
Parameter Driven Re-organisation
Parameter Driven Merging Policy
Grid File Directory Operations
Parameter Driven Splitting Policy
IDA Operations
Buffer Manager
Database Manager
(Virtual Storage)
E-Scales
Buffer Pool
R-module S-module data triples

Figure 2.8. The Triple Store architecture.

2.6 Discussion

There are several areas of the implementation just described where improvements could be made. These are now discussed.

2.6.1 String matching

String matching poses particular problems for the following reasons:

1. the recursive nature of functions makes them slow and difficult to optimise when executing string matching operations
2. the creation of unwanted string tokens during searches of text
3. the difficulty in handling case discrepancies and word contractions
4. the difficulty of coping with missing characters, quorum functions and word ordering.

Databases used in investigative systems will typically make extensive use of text and string variables and be used primarily in browsing mode [MAL96]. String matching can only be implemented by functions declared at the model level—such functions make use of three built-in base functions available as primitives: *length, substr* and *concat.* With these, other string manipulating functions can be constructed. In the current system text searching creates many unwanted tokens—this is a corollary of persistence and the way the search is recursively executed. A search through large texts frequently causes the available token space for strings to become exhausted thus giving unexpected results.

Displaying strings does not cause any noticeable delays because of clustering. However, each time a string is needed for a comparison operation, it has first to be re-assembled. This has proved a satisfactory method of string storage up to now because the majority of databases thus far have been accommodated in main memory. This situation would not be the case if much larger data sets were to be considered. There is also no garbage collection to remove old, unwanted strings from the store thus freeing up obsolete tokens for re-use.

With hindsight the handling of strings, and the importance attached to text searching, was not given enough priority in the original proposals. The reason for this is clear: all data were made to fit into the homogeneous triple store subsystem underpinning the data model. This restricts what can be efficiently done in terms of string matching. Furthermore searches that involve stem matching and elastic matching using wild-card characters, are harder to program at the model level. These areas are now recognised as weaknesses of the software and are discussed later in this thesis.

2.6.2 Tokenisation

The tokenising of integers and reals is trivial and sensible: it need not be discussed further here. However, the situation regarding string tokenising is not so clear cut.

There has always been a strong case for tokenising strings. Compact storage in memory (and on disk) together with ease of use by the compiler. Moreover, once a string has been tokenised, any repetition of the string (whatever length it may be) will only result in the addition of one extra 12-byte triple to represent it. These were the overriding reasons why a fully tokenised system was adopted.

However, tokenising strings implies a precision of data entry that is not always possible to achieve. Users can and do misspell words, they abbreviate them, use word contractions and varying formats of case. They may also want a search option where they only need enter a few characters in order for the search to be more general. A less rigid adherence to the concept of string tokens would allow for more scope when searching that takes into account imprecision and accommodates a greater flexibility for the user.

One of the achievements of TriStarp was the provision of persistence for all data. However, this gives rise to a conflict of interests between using it as a functional programming language in the traditional sense, and using it to support a database based on the functional model. Because of persistence a query posed, such as `length "George"`, would result in a token being created for George even though no data held in the database needs to be consulted. This is because comparisons are made between tokens. Moreover, a search for pattern `"Fred"` in text `"Christmas"` would produce tokens for `"Chri"`, `"hris"`, etc. as the search proceeded recursively through the text—even though the result of the expression would end up being false. A search through a text of length *t* for a pattern of length *p* could result in the creation of `t - p + 1` string tokens—the majority of which would be useless.

2.6.3 Functionality

There are two areas for discussion. Storage-level interface functions and model level functionality.

Storage level functionality

The decision to use a semantic-free triple store was in keeping with other research at the time [MAR84]; it was a deliberate attempt to keep the storage mechanism simple by providing a small set of interface functions. We suggest that the set of functions provided was perhaps too restrictive as it had to tie into the concept of a homogeneous triple store for all data. In order to provide optimisation techniques for string matching and parallel processing, we believe additional interface functions should be made available that would allow more searching and comparing tasks to be delegated to the storage sub-system where they can be handled more efficiently. This would allow data storage to reflect data usage in a way that helps boost performance in favour of text searching and browsing operations.

Model level functionality

Model level functionality was not discussed in much detail in the earlier sections of this chapter because it is largely a matter for each model level language developer to decide what to provide for their language. In the particular case of TriStarp, we have already highlighted the weaknesses of string matching and text searching options in section 2.6.1, and that more importance needs to be given to these areas to enable more efficient implementations to be developed.

2.6.4 The data model

From the outset, the TriStarp Group made a conscious decision to plan their research into database languages within the constraints of having to store their data at an atomic level as tokenised, semantic-free triples. The interface functions decided upon to provide access to the store placed further limitations as to how the data model could be manipulated, and drew a clear dividing line as to the demarcation of core functionality tasks such as join, search, etc. In other words, the storage and access mechanism came first and the development of the database languages at level 1—FDL, Exegesis [SMA88], Fudal [KIN92], Hydra [KIN96b] and Relief [MER98]—came second, and had to tailor their development taking the above considerations into account. However, the TriStarp work was done at an experimental level and not particularly aimed at a group of end users.

To say the design choices made were inappropriate would be wrong—they certainly provided a workable solution to the given problem at the time but, with hindsight, perhaps what was lacking was a mapping from the logical view of the world as triples to the physical view required for storage. The rationale behind the initial design choices were clearly in line with other research at the time [SHA88] but any attempts at optimisation now had to be done at the data model level (level 1) and were unable to take advantage of using additional, low-level processing power. One of the later additions to the TriStarp level 1 languages is Hydra [KIN96b] which makes a distinction between meta data and instance data, in that the former does not need to conform to the function graph model.

The way graph traversals are handled is that for each attribute A_{xn} linked by relation r_{xn} to entity E_x and used as a filter on instances of E_x, the required triples are accessed and collated sequentially in arriving at a final set of entity identifiers. Similarly, the final selection of attributes A_{yn} via relation r_{yn} from entity E_y is handled sequentially. The relevant segment of a list comprehension is below, then the figure shows the path taken.

`[(r`$_{y1}$` y, r`$_{y2}$` y, r`$_{y3}$` y)|| ... r`$_{x1}$` x & r`$_{x2}$` x & r`$_{x3}$` x ...]`

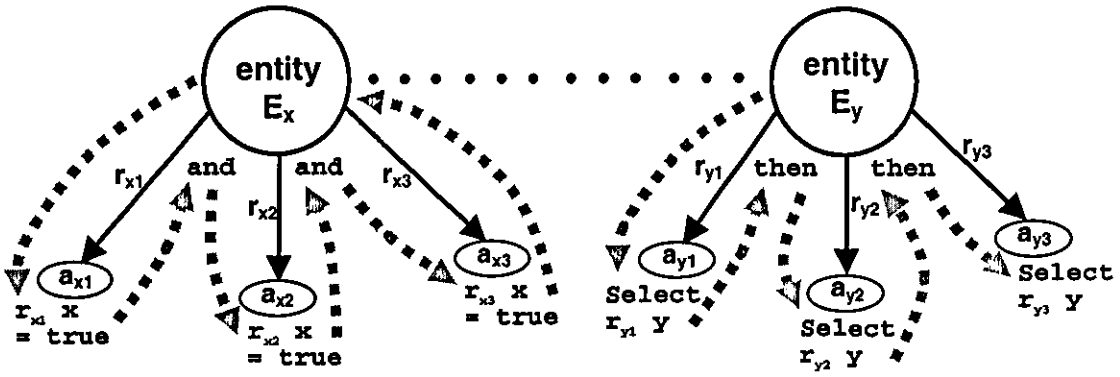


Figure 2.9. Graph traversal for attributes.

We believe that searching and collating attributes at the 'ends' of traversal paths can be handled more efficiently than this by using a different physical model and parallel processing techniques. These are discussed in later chapters.

As far as entity-entity graph traversal steps are concerned, the concept of a triple is a good one. The progression from *A* to *B* via relation *r* ($r{:}A \rightarrow B$) naturally fits the model of a triple <r, A, B>. However, a not insignificant amount of traversals are of the inverse variety ($r^{-1}{:}B \rightarrow A$) and these are not held as triples explicitly, they are derived through software. Searching for function inverses means constructing a temporary list of triples used to test for the inverse property. For example, the equation defining the inverse of function *f* is

```
        inv_f s <= [y || y <- All_t & (f y) = s]
```

which searches the extent of *t* to get a list of *y*, then checks that function *f* applied to each *y* evaluates to *s*. Although there would be additional storage requirements if inverse triples were to be held explicitly, we believe this can be achieved as part of a new architecture. Moreover, these triples could also be used to provide redundancy.

2.6.5 Directory structure

Directory organisation in grid files has been the subject of much research since 1984. There is no doubt that the original grid file directory has been improved upon by the data structures used in the BTM. This is as a direct result of using a *select_sp_point* algorithm to reduce the number of grid blocks, together with the IDA data structures to reduce the number of data page pointers. However, the efficiency of using the IDA can vary greatly according to the configuration of data space regions and the high likelihood of a non-uniform distribution of data. Of the nine databases analysed in [DER89], one configuration resulted in 11 page accesses being necessary, although the total directory size (S-module plus R-module) can be a linear function equal to the size of non-uniform data distribution.

The concept of a three-dimensional grid file is a good generic model when storing database records that have a fixed format for their three, key attributes. Earlier attempts at storing triples in hierarchical structures, such as B^+-trees [DER85], proved unacceptable as it was difficult to add a re-tuning facility and gave poor performance of response time for partial match queries at lower levels of the hierarchy—although there was efficient space utilisation. Therefore, a grid file variation seems to remain the best method for indexing records of three key attributes, if that is to remain as the underlying data model.

One of the reasons for adoption of the grid file was that it easily lends itself to the indexing of triples, which could now be accessed from one or two of the three fields. The grid file is also a dynamic indexing structure and the implementation used for the triple store can have its parameters set to give a clustering priority to a particular dimension. However, frequent updates and re-organisations with splitting or merging of pages have to be made to maintain a balanced index. Access via the third field of a triple (the object) can cause lengthy delays if the parameters are set to favour clustering on the relation or subject fields as is often the case. Even if a later session with the same database file re-sets the access priorities in favour, say, of the object and object-relation combination, the previously-entered data will still be structured around the parameters supplied when it was loaded.

It was found in the BTM implementation that frequent re-organisation of the directory—together with the splitting and merging of pages—led to a less efficient system than was expected [DER89]. The storage and searching of text does not necessarily lend itself to this highly idealised form of indexing and would perhaps benefit from a coarser indexing structure and greater processing power, provided at a lower level, to facilitate the flexible searching of loosely structured text.

2.7 Summary

In this chapter we have reviewed the storage architecture underpinning the TriStarp software and identified areas for improvement, paying particular attention to string matching and graph traversal operations. Tokenisation is limiting what can be done to broaden text-searching capabilities and boost overall performance levels. Interface functionality needs to be enhanced so there is more choice available to users. Finally, we suggested that a mapping from logical triples to physical records was lacking in the original proposals and that not all data need conform to the function graph model. A less dynamic approach to indexing would be worth investigating too.

In the next chapter we begin by examining alternative variations to the triple store. We then discuss the categories and needs of users before reviewing storage and access methods in general. Finally we introduce our alternative to the homogeneous triple store architecture outlining the areas for further investigation that are discussed in the later chapters.

Chapter 3 Background to areas of work covered

3.1 Introduction

As set out in chapter 1, there are several areas of work described in this thesis. This chapter provides background material to these areas as an introduction to the individual topics of interest covered in later chapters. The main items presented in this chapter are therefore: a review of other grid file and binary relational storage structures, an introduction to user requirements and text searching, a brief review of storage and access methods, using a search engine and parallel processing. Following these topics, our proposals are introduced which then form the remainder of the thesis.

3.2 A review of similar alternatives

We describe the evolution of grid files since their introduction in 1984, then highlight more general binary relational storage structures whose concepts have been around for much longer. Finally, we consider the latest triple store implementation that involves the use of space-filling curves and draw conclusions for this section.

3.2.1 Grid file variants

Since Nievergelt's keynote paper much research has been done arising from the original proposals for the grid file [NIE84] and several hybrid systems have been developed [HIN85, WHA85, WHA91, OUK85, OZK85]. (See [DER89] for full details of these schemes.) Since then research has continued investigating ways of providing optimisations to the original grid file design. Ouksel *et al* have improved on their Interpolation-based grid file in relation to concurrency control [OUK92, OUK94]. Whang and Krishnamurthy *et al* have continued their research into the Multilevel grid file improving the execution of join operations [KIM95] and directory growth [KIM97, KIM98]. These schemes have one thing in common: they all use a cyclic, halving of domains in alternate dimensions when splitting data space regions which must remain hyper-rectangular at all times.

The one notable exception is the BANG file [FRE87, FRE89]. The BANG file allows nesting of data space regions and so a hyper-rectangular shape is not necessarily required.

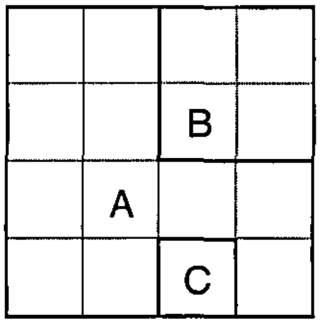


Figure 3.1. The BANG file scheme.

The distinctive feature to note here is that embedding of data space regions is allowed, and the way this embedding is handled. In this example, data space regions B and C are embedded in A. So, to obtain the records that are in the logical data space region A, a subtraction of the data space regions embedded in the physical region A must be done—thus, A-B-C—which gives the appropriate data pages to search.

As well as nesting of data space regions, the BANG file uses a splitting algorithm that will select an embedded area in the over-populated region such that there is an even distribution of records between the two logical regions. The BANG file also has a directory size that is a linear function of the number of stored records whatever the record distribution. This is because it avoids the creation of empty regions when splitting, and has a single entry in its directory for each data page. The BANG file also enjoys a higher degree of freedom when merging regions, since a region can merge with either: the region in which it is embedded; any region which is embedded in it; or its buddy region. Since then, Hosur *et al* [HOS92] have improved upon the established superiority of the BANG file, by providing efficient, dynamic adding and removal of attributes which results in changes to the dimensionality of the structure—although there are serious shortcomings with the index construction and maintenance. Moreover, a full implementation of the BANG file has never been built [LAW00].

There is another partitioning method for grid file directories [CHU89] which is worthy of note as it is similar to the *select_sp_point* algorithm used in BTM. This system improves an earlier algorithm by Cranston [CRA75] that guarantees non-empty data space regions. Some other recent work on grid files includes a spatial grid file for multimedia data that is specific to high dimensional data indexing [ALP97], and a novel scheme to handle temporal interval data (as well as other attributes) in a way that does not result in a skewed directory structure [LEE98]. In this scheme, the data space is represented by a right-angled triangular space so that the time start (TS) is always before the time end (TE).

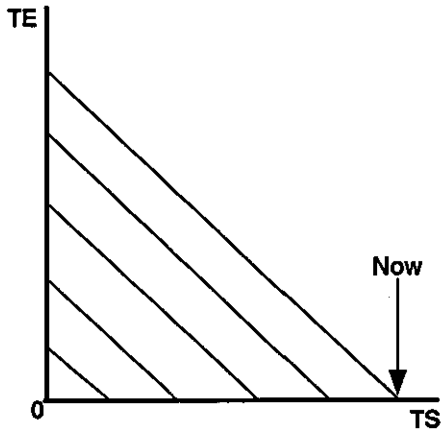


Figure 3.2. The temporal grid file.

The valid grid space is the lower-left section shown in Figure 3.2. Records are always inserted at point Now. As time passes the hypotenuse moves up and out from the origin 0 ensuring TS ≤ TE.

The main design features of the grid file are as follows. For a reasonably large relation, retrieval of a tuple requires at most two disk accesses—one to the correct portion of the directory and another to the correct data page that holds the tuple. The nature of the file structure is order preserving on each attribute domain, so that tuples that are close logically are likely to be close physically. These properties allow for efficient retrieval of point queries and range queries. In theory, the grid file index structure can adapt gracefully to insertions and deletions and performs well in static or dynamic operations although this is not always the case.

3.2.2 Other binary relational storage structures

There have been several implementations of binary relational storage structures over many years. They include the following early variants—discussed in detail in [DER89]—plus some more recent additions.

Relational Data File (RDF) [LEV67] which is quadruple based and holds four separate files for data—each one indexed on one of the four key attributes.

Leap [FEL69], which uses triples and holds three copies of the data—each one indexed by hashing on two of the three attributes. As static hashing is used the file size must be estimated beforehand and, after overloading, performance degenerates rapidly.

Titman [TIT74] proposed a **triple-based system** where a separate file was held for each relation in the database. In each file the `<subject, object>` pairs would be stored. This results in efficient storage but poor performance where the relation field was not specified. A very similar method is used by the Well system [MUN78].

The **FACT** system [MAG80, MAG82] uses a quadruple approach with a look-up table maintained for conversion of external attributes into internal, fixed-length tokens. Entries can be arbitrarily nested and generalisation is supported. There are three lists maintained for every internal identifier—one each for where the identifier appears as relation, subject and object. This concept has been followed up more recently in what has been termed the **Associative Model of Data** [WIL00]. This is discussed at greater length later in the thesis.

NDB [SHA78, WIN79, SHA88] adopts a triple approach and represents every entity with a three-component data structure called a v-element which holds links to the other elements linked to the current one. There is no distinction between entities and attributes, as they are all stored in the same way.

The **Oggetto** object-oriented database system [MAR92] uses a BRSS to underpin the storage of facts and methods. It includes inheritance and allows for object migration and database closure (via an *expand* function). There are three triple stores used for access:

- a triple 'heap' that is used for temporary storage of triples
- Trible—a system that uses inverted files to speed access
- OSROS—a system that uses combinations of the three fields for dynamic hashing of triples.

Two other points of interest are that in-store object information is held in a data structure that stores the name and type of every attribute in the database. This structure also holds details of instances of a type, which are built up internally before being stored as triples. They also agree that names (strings) are better stored away from a homogeneous triple store and it is this store that is used to supply the identifiers that are then used in the main triple store. This idea was used in the **Universal Triple Machine** [SHA88].

A more recent adaptation of a binary relational storage structure is the **3-tuple model** as presented in [OZA96]. The idea is that users are allowed to store data first and then structure the data via a schema. The relationships between schema and data, and schema and schema can be altered without changing the whole database and is termed the 'bottom up' approach. Each tuple is of the form `<object, attribute, value>` and can accommodate arbitrary nesting. An n-order, finite, directed labelled, graph is used to represent objects. A set of tuples with the same first element represents an object; a set of tuples with the same second element represents an association. In the model, instance data and meta data are unified in the graph by object classification. Basic objects are integers, reals, etc. Compound objects are further divided into three types:

- class object – represents a set of instance objects which share the same property—i.e. it is a schema
- instance object – holds the data for each class as above
- free object – is not constrained by either of the two class objects above.

There are also class relations, instance relations and free relations to match the above objects. Free objects are able to migrate from one schema to another but it is not clear how this migration is handled at the physical level—nor is it clear how data is stored physically in any case. There is no temporal dimensionality discussed, nor is an indexing structure mentioned.

3.2.3 A triple store based on space-filling curves

A recent addition to TriStarp has been a triple store based on space-filling curves. The concept of space filling-curves has been around for a long time [HIL1891] and only a simple example will be given here—for more details see [SAG94]. The basic idea is to map an n-dimensional space on to a one-dimensional linear array so that adjacent points in the n-space are as adjacent as possible in the array.

Taking the case of the Hilbert curve as an example this is best described pictorially (in two dimensions for clarity) in Figure 3.3 below. Several curves were investigated but the Hilbert curve proved the most successful. This is because it has the adjacency property of being at all times continuous—whereas the other curves do not share this property.

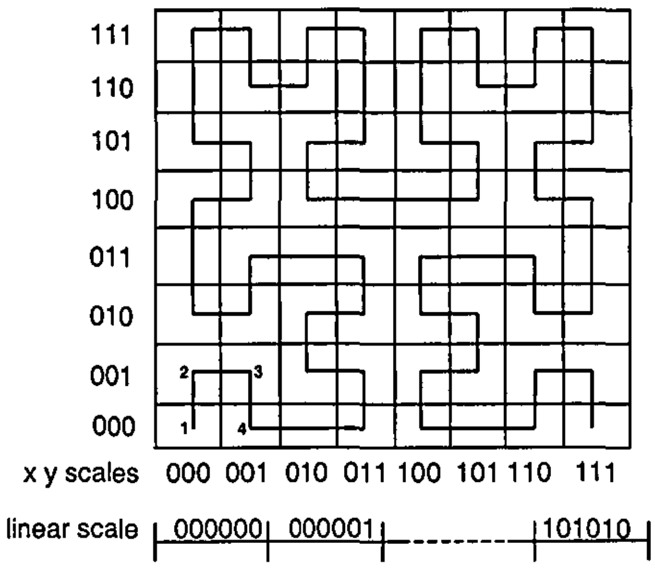


Figure 3.3. The Hilbert Curve

The linear scales are formed by concatenating the x and y co-ordinates for the whole of the data space. In this example, there are 64 points in the two dimensional data space to be mapped onto 64 partitions of the array. The curve starts at the bottom left of the figure (datum point 000000) and follows the line until it reaches the bottom right which has datum point number 111000. The datum points do not map exactly to the numerical partitions of the grid. For example, the third datum point will have partition number 3 (000011) but the co-ordinates of the curve will be 001001 (x concatenated to y).

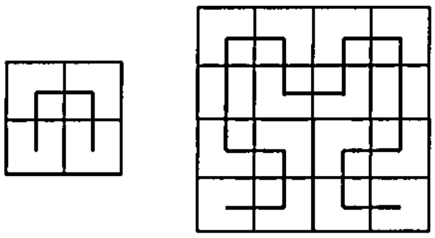


Figure 3.4. First and second order curves.

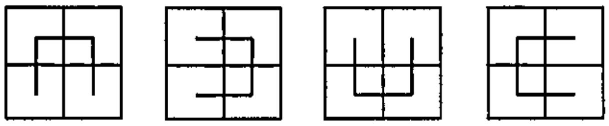


Figure 3.5. States the Hilbert curve can take.

Moreover, the number of steps or iterations of the curve—called orders—(Figure 3.3 is a third-order curve and Figure 3.4 shows first and second order curves) require different orientations (states) for the direction the curve takes as shown in Figure 3.5. This is to ensure it is at all times continuous thereby retaining the adjacency property. There are therefore four orientations of a curve section. These are referred to as states—in the state diagram sense—and by knowing the state and the order of the curve an accurate key value can be obtained. An important difference from grid files is that the splitting and merging of pages is made according to partitions of data rather than partitions of the key space. This approach obviates the problems of partitions overlapping within the index.

The array is used to construct, algorithmically, the appropriate data pages to search, which are held in a B-tree structure that can expand and contract as necessary. More details and a full evaluation of this scheme are given in the PhD thesis of Lawder [LAW00]. An immediate observation is that the method of page searching could be amenable to parallel processing techniques. If a page is in a block that contains (say) four smaller blocks, then a search of these four blocks could be done in parallel. This is an area for further investigation.

Initial results described by Lawder [LAW00] indicate that a space-filling curve triple store performs similarly to the current implementation based on the grid file in two or three dimensions. However, as the number of dimensions increases, the space-filling curve triple store outperforms the grid file triple store. An issue not addressed by this approach is the complexities involved in the relational algebra: join, intersection and union of large sets of data. Moreover, the experiments were run against data sets held entirely in main memory; in large, practical applications it is most unlikely that all data would be held in memory.

3.2.4 Conclusions for grid file and BRSS applicability

Frost identified the advantages of binary relational storage structures as providing a simplification of system design and use and, improvements in data independence. He also cites the disadvantage that data may only be retrieved singularly—groups of related items may only be retrieved by issuing several commands. Added to this is the fact that the majority of triples are ordered on the relation field, so are likely to be clustered on that field in preference to any common entity they have.

In his original proposal for the storage of binary relations, Frost suggested holding six copies of the triples, with each set indexed on one of the six simple associative forms for querying the database. This is the ultimate fast access solution but has the most serious implications for update out of all the systems described. The trade-off is often one between speed of access and cost of updates. However, there is often little to choose between many storage methods that use either triples and/or fixed-length tokens to store data. As a modelling concept a triple is a good idea. But, from a storage viewpoint the idea has never caught on in the same way and is too restrictive to accommodate easily the richer data types that are now required.

3.3 Text searching and user requirements

In this section we briefly introduce the fundamentals of text searching and user requirements. Readers familiar with the background to these areas may wish to proceed directly to section 3.4.

3.3.1 An introduction to text searching

Text searching is a vast subject area alone and is only covered briefly here. For a more detailed explanation the reader is referred to [MEA92].

A text can consist of words, collections of words (clauses and phrases), sentences, paragraphs etc, all comprising alphanumeric characters drawn from the domain of the grammar. Differentiating between these classes can be difficult and often depends on the meaning of the text (its context) and any delimiters used to break up the text into smaller, multiple word structures. For instance, is "bogus gas-man" a text, phrase, two words or three words? The choice can vary depending on what the collection of characters will be used for. At the lowest level a word might be considered as a collection of alphanumeric characters delimited by white space characters (single space, tab, carriage return etc.) The choice of word delimiter can be crucial and, although this is usually the space character, it need not be.

The degree of freedom for searching must be greater for text than for words with multiple word structures somewhere between the two. The same words in a text can be analysed in terms of various patterns of occurrence. A text has vocabulary—but this need not be tightly controlled—as well as patterns of vocabulary where syntax can indicate word ordering for example. The patterns can be used to construct an index of terms for searching.

The actual form a query takes is often quite simple and is likely to have the format: *attribute condition value*—as in: `"author = Shakespeare"`. This fits naturally into the triple construct. Variations on this can include provision for truncation of the search pattern, inclusion of wild card characters that can represent one missing character or zero or more missing characters—whether in words only or across word boundaries. Case sensitivity can easily be accommodated too. Another feature of text searching is the ability to make proximity searches. In this case, word ordering and space between words can be passed to the search algorithm via meta characters included as part of the search pattern or indicated in some other way. This can be extended in various ways

Stem matching is important in text searching as it allows different inflections from the same base word to be located. Thus, if the search was for "harmony" and any derivatives, the pattern entered could be "harmon%" where "%" indicates match zero or more characters to the end of the word. This search might find harmony, harmonise, harmonious, etc. It would also find harmonica, which, in this instance, would not be required. But, it is often better to have terms returned that can be accepted or rejected at the discretion of the user than allow the system to make such judgements. Stem matching allows the users to enter short strings quickly and be used in conjunction with various search terms over the same text.

To search for a person whose name is Fred or Frederick, who lives in Lower something or other and whose job is something to do with telecommunications, the search could take the format:

```
fname = "fred%" ∧ street = "lowe%" ∧ job = "tele%".
```

This is also the way British Railways ticket machines operate and British Telecommunications inquiries are handled.

There are other text retrieval techniques that construct (quite complex) indexes for the majority of words in the text. The excluded words are known as stop words and include the most often-used words in the English language—prepositions, articles, conjunctions and pronouns. Stop words are a list of very general words that lend no significance to identifying the subject matter of a text. In [MEA92] the stop words are given as: *an, and, by, for, from, of, the, to* and *with*. These can be viewed as a base to which other words would be added depending upon the context of the application domain. (Note that the letter A is not included; this can often form a search term as, for example, it represents a vitamin.)

The maintenance overhead and the sheer size of many index files (which can be bigger than the original text) often supports the argument for not making heavy use of indexes. Other options include: similarity measures that make further statistical pre-processing knowledge available about the text for use in searching; text or association techniques that use word occurrence statistics to measure the strength of association and; clustering techniques that group together records that have similar frequency distributions for attribute values. The use of signature values that act as (shorter) keys for frequent search terms is another optimisation that can be used. All these techniques involve the use of much meta data that can be cumbersome to maintain and can negate the benefit of its use.

3.3.2 What users expect

In this short section, unless otherwise specified, the term *users* implies regular users, such as data query and data entry operatives as opposed to systems personnel. We quote material from Southerden specifically, as he outlined an improved interface for ICL's well established investigative systems software—INDEPOL [SOU97]—which is very relevant to our area of work.

Users want a simple view of their data. This means that the view offered by the database system must, as realistically as possible, reflect the general conceptual view of data they might have in a non-computerised setting. They do not want, nor need, to know about data mapping, storage sub-system structures, indexes, etc.

Users want to express their needs simply. Users are likely to be skilled in the use of the computer system and have a good working knowledge of the database—its content and structure. They want to express their needs with the minimum of typing effort. To satisfy these needs the database management system should provide an enquiry update language with such tools as form filling, palette provision, drag and drop facilities and a syntax-directed editor—all in a graphical user interface environment. To a certain extent the TriStarp Group has achieved this as their software includes a graphical query language, Gql [PAP95]. However, this area is still the subject of continuing research.

Users expect a fast response. The time that elapses between a user issuing a search command and getting the answer back is crucial. It's difficult for a user to switch attention while waiting for the outcome of a task and any delay is seen as lost or unproductive time. When using a system in browsing mode, users often want to build up a search query by increasingly refining their search parameters to hone in on their target. Coupled with this is the reality that users may have to make several, related queries to a database before obtaining anything useful. The data management program should have the ability to recognise requests that will take a long time to service. Such information should be relayed to the user who then has the choice of whether to proceed with the request or to abort it.

Users expect protection against misuse. The sorts of misuse implied here are security of access rights etc, and integrity of data that may be lost or rendered unusable during updating.

3.3.3 Categories of users

Southerden further identified four core sets of users:

1. those whose main tasks are data entry and data query
2. those whose main tasks involve simple investigations, research or analysis
3. experts who build queries for others and
4. technical developers and support staff.

Of these the vast majority of users fall into the first category. INDEPOL is a tried and tested investigative system, so it is sensible to adopt the same user priorities and concepts. We can classify the above user groups into the following categories including a split of the first category:

1. data entry operatives
2. data query operatives
3. experts who build search strategies for category 2 users and
4. database developers and support staff.

Data entry can be done in two ways: bulk loading at the time the original database is created and on-line by users with pre-defined forms—which is another feature used in INDEPOL Client. The experts who build the meta functions in category 3 will need to be aware of any schema updates, integrity constraints etc., to build sound search macros and functions. Category 4 users are the database administrator and development staff responsible for (among other things) schema evolution, updating the database, garbage collection, security and integrity issues.

3.4 A review of storage and access methods

Following our review of grid files and binary relational storage structures earlier in the chapter, we now address the wider aspect of storage and access issues. Basic file structure design is briefly covered as well as the argument for main memory databases. Finally a summary of design considerations is presented.

3.4.1 File structure design issues

Here we discuss basic file structure design issues and compare and contrast them. Our arguments for our chosen architecture will be given later in this chapter and expanded upon in the following chapters.

The overall aim of the storage sub-system in a large, operational database system is to arrange the data in a suitable format—across one or more files if necessary—in such a way that data can be accessed by the application programs that need to use it in an efficient way, while maintaining security and integrity considerations. At a fundamental level, data is traditionally held in atomic structures that can be built up into data structures like records, sets, lists, etc. of the required complexity for the application. The first design issues to consider might prompt the following sequence of questions:

- what do we want to do with the data? Mainly browse it or alter it?
- who is going to access it and what views do they want of it?
- what are the types of the data to be stored (text, numbers, etc.)?
- how should the data be ordered to effect the required accesses?

The last question usually comes down to a choice between single-key or multi-key ordering. For single-key processing only one attribute of a record is used to order the file and the main access choices are to use indexing methods based on B-tree and hashing. Hashing is fast if only one record is required—but not so suitable for range queries. Btrees are more appropriate for retrieving a range of records.

If records are to be retrieved by more than one key then the multi-key file organisations are more suitable. Access methods for these structures often involve a complex indexing structure using inverted files that can add a considerable overhead when updating or adding records. The more recent grid file organisations, mentioned earlier, are a more natural way to index multi-key record structures as they can guarantee a hit in no more that two disk accesses for known information. Also, the directory used in grid files can adapt more gracefully to record insertions and deletions; range queries are also well supported.

In general, determining the best file organisation method and the most efficient access techniques are difficult. We consider the basic parameters that can be used to determine the best method to be as follows: *time, file-use ratio, space,* and *volatility* (from [SMI87]):

The ***time*** parameter includes time to develop and maintain the software: the more complex the file structures required, the greater the time factor for updating the storage sub-system. If updates have to be done off-line in batch mode, then this must be included in the time parameter.

The ***File-use ratio*** is obtained by dividing the total records held in the file by the number of records actually used. If the ratio is high, meaning that most of the records are needed regularly, then a sequential organisation method is preferable. If the ratio is low a hashing scheme might be better.

The ***space*** parameter refers to the total space requirement for the instance data, meta data, indexes etc. that are associated with the system. Any space needed for temporary sorting or other re-organisation of the data must also be included. A balance has to be struck between what is to be held on disk and what needs to be held in memory (see the next sub-section for further discussion on this). How are updates to be handled? What space is needed for recovery procedures in case of file loss or corruption of data? What can be held in duplicate to speed up access?

Volatility concerns how often the data held in the file changes. If there are frequent changes to the data the maintenance of complex access structures like indexes or having to re-hash some of the data may prove unworkable. On the other hand if most of the data will remain unaltered, even though a lot of it may need frequent accessing, some form of coarse index-sequential access method might suffice.

An important aspect affecting the design choice concerns maintaining database integrity. The well-known problems of update anomalies across files, tables or data structures in general are, unfortunately, still with us. Also, an integrated file system must handle concurrent access (where applicable) and maintain data integrity. Other more fundamental design issues include:

- selection of page size (affected by logical record size)
- selection of blocking factor (number of pages per block)
- allocation of buffers (multiple buffers can significantly improve performance)
- organisation of blocks on secondary storage
- handling of file growth (static or dynamic) and
- reorganisation point (a point where a thorough reorganisation of the files is required so that performance does not deteriorate beyond acceptable levels).

Finally, a choice has to be made about whether the files should be static or dynamic. The above four criteria—*time, file-use, space* and *volatility*—clearly play an important part in this choice but there are other difficulties surrounding dynamic file organisation. Held and Stonebraker [HEL78] suggest that, although a dynamic index is easier to maintain *in situ*, the cost of doing so is threefold: insertions, deletions and movements within a B-tree can result in complex pointer maintenance; concurrency problems can occur—locking out a B-tree node is non-trivial and; additional pointers are required in non-leaf nodes because they can split/merge dynamically. The branching factor is thus smaller and the height of a tree likely to be greater than that of a comparable static index. Operations such as search, insert and delete will therefore take longer than for a static structure with no overflows.

3.4.2 Main memory databases or disk-based databases?

Up until now TriStarp databases have fitted into main memory. The arguments for main memory databases are forceful [GAR92] and frequently include some form of encoding (tokenisation) coupled with increased solid-state memory [COC98]. However, there are drawbacks to this philosophy, which include the volatility of main memory to failure, resulting in the need to make frequent back-ups to disk anyway. Moreover, there will always be databases that are too large to fit into main memory; the requirements of growing data sets matches the improvement in main memory capacity, and this trend is likely to continue. Important factors to consider are set out below:

- main memory access costs are orders of magnitude less than disk-based access costs
- main memory is often volatile, whereas disks are non-volatile
- disk accesses have fixed costs for blocks, while main memory is not block oriented so costs are variable
- the layout of data is crucial on disk but not in main memory

- sequential access is faster for disk than random access. Sequential access is not so important for main memory
- main memory is directly accessible by the processor, while disks are not.

Whether or not to use a main memory database system or a disk-based system ultimately comes down to the specific application domain that is required in each case. There is a compromise for some large databases whereby data can be designated 'hot' or 'cold'. Hot data are often accessed or modified: Cold data are less often accessed or modified. The former can be held in main memory: the latter on disk. The idea of splitting data this way seems complicated for a text-intensive, application domain that would not facilitate searching operations in the optimum way.

3.4.3 Summary of design considerations

In this section we summarise the salient design issues discussed earlier setting out how they best fit our specific requirements. The specification of our database system has the following characteristics:

- it will be used mainly in browsing/searching mode
- the system will be used to link facts across a function graph model
- data hypothesis is the responsibility of the user, not the system
- much of the data will comprise strings, some in large texts
- it is not imperative that all data be updated dynamically, some updates can be performed off-line†
- information will, at the lowest level, be represented by atomic binary relationship between entities and attributes (non-lexicals and lexemes)
- there is a close relationship between attributes of a common entity

† The UK Inland Revenue name and address file holds 48 million records and requires the addition or amendment of around 5% of these on a daily basis [WIL85].

- there is often a close relationship between attributes shared by different entities
- file-use ratio will be high because of the searching and browsing operations
- space saving is not a crucial factor. However, some removal of duplication is desirable where this does not impair performance
- because a functional paradigm is used, the properties of referential transparency guarantee freedom from undesirable side effects.
- integrity constraints, at the meta data level, are already a successful method of enforcing database integrity
- it is most likely there will be too much data for a main memory database system to be used
- the system is to be multi-user.

From our specification above we believe the bulk of the instance data should be stored on disk in a format well suited to rapid searching techniques while leaving main memory free for other uses. It would be difficult to upgrade a main memory system to a multi-user, browser-oriented, client-server environment—qualities that a large, text intensive, operational database system would need to have. If instance data is kept primarily on disk in a client-server environment, it is easily accessed by all system users for browsing etc, while leaving reliability, integrity and security as server system functions [PRO98].

3.5 Using a search engine

The incorporation of a search engine as a data filter on each processor forms part of our architecture proposals so is introduced here. Readers familiar with the concepts behind search engines may wish to skip this section and go to section 3.6.

Search engines have been in use for twenty years now and are therefore considered 'mature' technology. However, they are still a useful tool and have an established track record where devolving certain searching and data filtering operations are concerned—although there are some architectures that search engines are not as capable of exploiting as others. Some design considerations—such as record structure—are therefore crucial in making a decision to use a search engine. There have been several search engines used over the years—a good résumé is given in [SU88]. Here we describe the basic concepts behind one such search engine that has been in commercial use for over 20 years.

The ICL Content Addressable Filestore (CAFS) [MIT76] is connected to a disk controller and accepts logical requests for data. The disk controller has hardware that can perform key matches. The controller can be loaded with constants to describe record fields and values and with a microprogram to determine if a particular record satisfies a request. See [CAF85] for various papers describing the components of CAFS. A good general description of CAFS can be found in [MAL79] and a brief overview of the key components is now given with the aid of Figure 3.6 found in [BAB79].

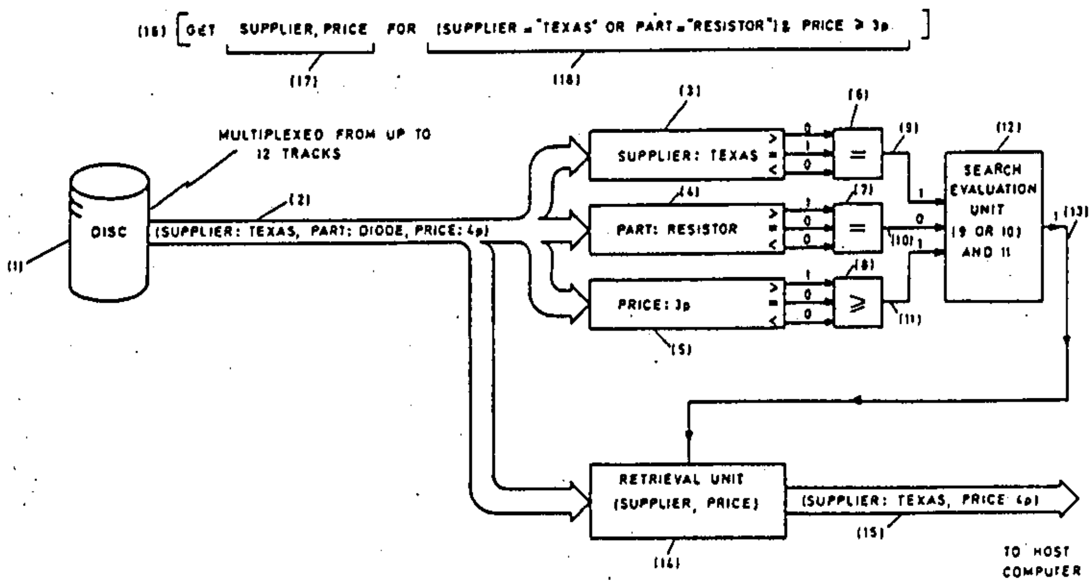

(16) [GET SUPPLIER, PRICE FOR (SUPPLIER = 'TEXAS' OR PART = 'RESISTOR') & PRICE ≥ 3p]
(17)
(18)
MULTIPLEXED FROM UP TO 12 TRACKS
(1)
DISC
(2)
(SUPPLIER: TEXAS, PART: DIODE, PRICE: 4p)
(3)
SUPPLIER: TEXAS
(4)
PART: RESISTOR
PRICE: 3p
(5)
(6)
(7)
(8)
(9)
(10)
(11)
(12)
SEARCH EVALUATION UNIT (9 OR 10) AND 11
(13)
RETRIEVAL UNIT (SUPPLIER, PRICE)
(14)
(SUPPLIER: TEXAS, PRICE 4p)
(15)
TO HOST COMPUTER

Figure 3.6. Outline of CAFS.

The selector (18) sets up the key registers (3-5) with the key value pairs taken from the selector. The tuple (2) arrives from the disk (1) and is placed in each key register where a latch is set according to the theta condition. These are compared to the required theta condition and the latch comparators (6-8) are set before the result is passed to the search evaluation unit (12). Here they form part of a logic expression derived from the selector that, if true, informs the retrieval unit (14) to pass the relevant items from the tuple to the host computer. The search evaluation unit (12) can handle nested Boolean expressions and threshold functions and the hardware can support up to twelve disk channels via multiplexing.

3.5.1 Record structure

To facilitate the use of a search engine the data must be stored in a format that permits the various filtration processes. The options CAFS offers include: logic operators AND, OR and NOT; weighted threshold functions; theta conditions =, ≠, >, <, ≥ and ≤; masking of data items to byte level and stem matching. The record structure has therefore to include data and field identifiers as well as the actual data. This can be done to various levels of granularity and is introduced in chapter 4 where it is highly relevant to document structure where this mechanism is required. Typical record structure is shown below [MAL79].

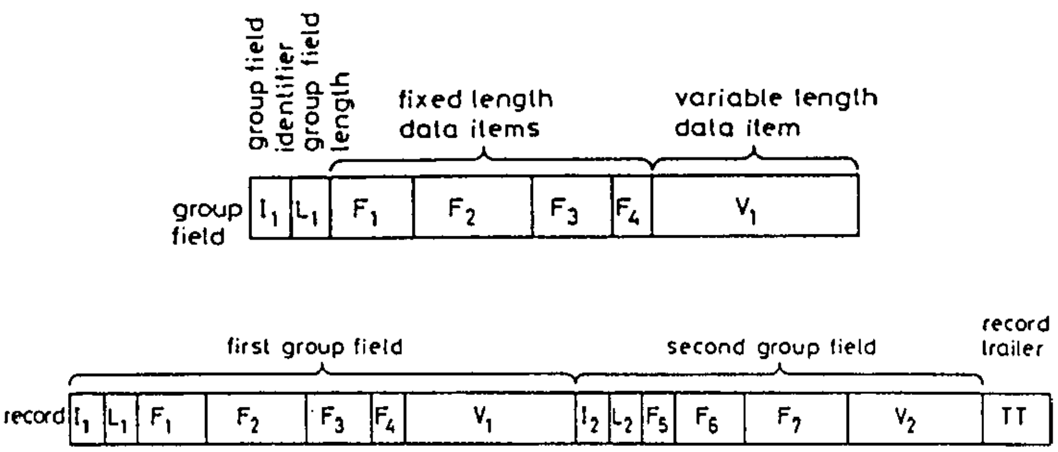

group field identifier
group field length
fixed length data items
variable length data item
group field
I_1 L_1 F_1 F_2 F_3 F_4 V_1
first group field
second group field
record trailer
record
I_1 L_1 F_1 F_2 F_3 F_4 V_1 I_2 L_2 F_5 F_6 F_7 V_2 TT

Figure 3.7. Typical record format.

For the key registers to carry out their function, the data needs to be stored in fixed field format or permit variable length fields with identifiers included (as shown above). If an individual item is required for comparison, e.g. stem matching, it can be isolated easily by using a mask. Using CAFS as described in this section will typically decrease search times by a factor of between 5 and 100. Moreover, the workload for the processor is reduced by more than 90% as the amount of data transferred to the processor is significantly less than in traditional methods when considering un-indexed data. Three different implementations of the CAFS product have been achieved in three technologies. The latest implementation being for VLSI in the late 1980s [ILL96].

3.6 Parallel processing

In this section we again introduce the basic concepts behind parallel processing, so readers familiar with these may wish to proceed to a later section.

3.6.1 Implementing problems in parallel

A problem may be solved by exploiting the parallelism inherent in an algorithm (algorithmic decomposition) or by applying the algorithm to different parts of the problem (domain decomposition). Domain decomposition involves examining the problem domain to ascertain the parallelism that may be exploited by applying the algorithm to several distinct sets of the data at the same time. The solution can be applied as a *data driven* method—where the sum of the tasks is divided into the number of processors used, or *demand driven* method—where each processor 'demands' a new task from a central 'pool' as and when it finishes a previous task.

A processor taxonomy was established by Flynn [FLY72] based on principal interaction patterns of instructions and data streams. The most useful of these has proved to be *multiple instruction, multiple data* (MIMD) where each processing element (PE) can operate asynchronously. By providing the processors with the ability to communicate with each other, they may interact and co-operate in the solution of a given problem. The level of interaction and access to memory has led to two types of system being developed. Where global memory is shared the interaction is known as *tightly coupled* and where each PE is responsible for a section of (private) memory the interaction is known as *loosely coupled*. These are shown below.

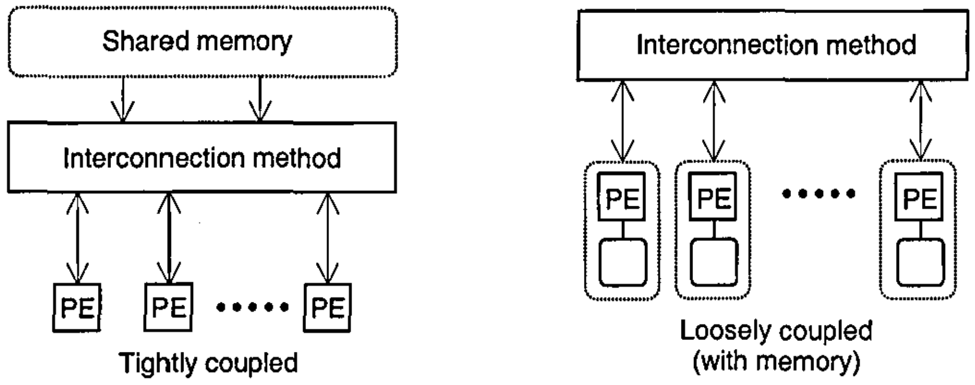


Figure 3.8. MIMD configuration.

A *topology* is a number of processors connected by a network in some configuration—token ring etc. A *process* is a segment of code that runs concurrently with other processes on a single *processor*. A *processing element* (PE) consists of a set of processes used in harmony on a single processor. Links in the interconnected system can be between processes on the same processor (internal links) or between processes on different processors (external links).

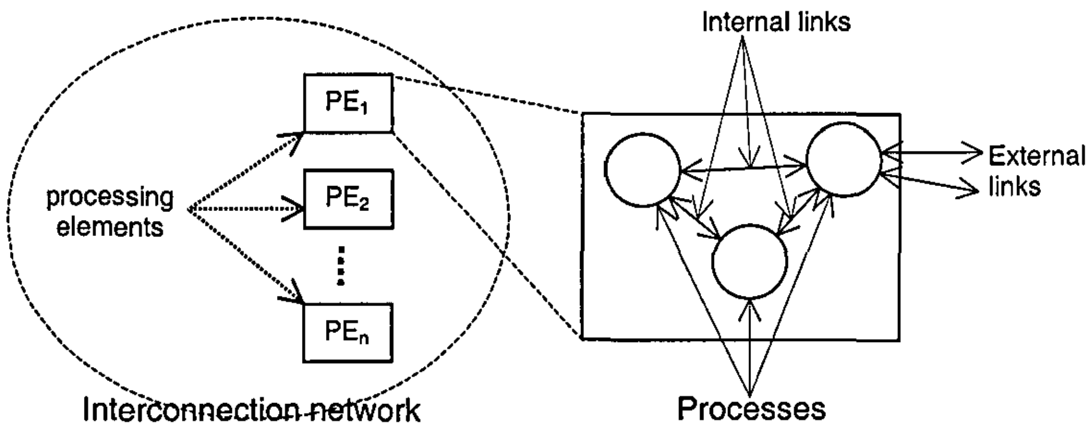


Figure 3.9. Parallel processing terminology.

To provide a useful parallel processing environment there must be access to input/output facilities. It is customary to achieve this by using one of the PEs as a system controller. As well as handling the input/output interface, the system controller is responsible for collecting and collating the results from the other PEs.

If PEs could spend 100% of their time doing useful computation, linear speed-up would be possible and each PE added would improve performance. In practice this is not possible. The choice of an appropriate computational model is of paramount importance to ensure that each data item is acted upon and determines how tasks are allocated between PEs. The optimum model will see that the workload is distributed evenly among all available PEs. The computational models are

Data driven—where data items are allocated to PEs in advance of computation. It can thus be considered a 'static' scheme where the computational requirements of data items must be known in advance.

Demand driven—the opposite of data driven and considered 'dynamic'. In this model work is allocated to PEs as they become idle. This model comes with greater communication overheads but problems like 'load balancing' are more easily handled.

Hybrid—a combination of the above two and useful where an initial set of known problems can be handled statically before other problems of unknown complexity are tackled more dynamically, depending on the demands of the situation.

If the size of the problem domain is too large to be accommodated in its entirety at one PE then it may be distributed across all PEs as well as secondary storage devices if required. The management of the data involves optimising data fetching and use of cache at PEs to maximise the solution of a problem.

3.6.2 The choices for parallel implementation

Parallel processing of functional languages has been a research activity for several years. The original hopes for exploitation of the implicit parallelism in functional languages led to several avenues of research in a number of promising projects. However, the task of implementing a parallel functional language is much more substantial than it first appears [TRI96]. Difficulties include the management overheads involved and the increased complexity for the operating system. Many parameters for parallel tasks are of a dynamic nature, which adds further complexity. Wilhelm [WIL96] in fact argues that the difficulties of parallel execution are likely to remain. A good résumé of early research is given in [LIN96] but some notable areas are highlighted here.

FAD, implemented on the parallel database machine **Bubba** [BOR90], did not incorporate list comprehensions instead using operations like *map* and *filter* to achieve the same results. Moreover, in FAD functions were not first class objects and updates were imperative.

GRIP is a functional database implemented on a parallel machine [PEY87a]. It does not have a parallel I/O system so the database has to reside in main memory. It is also based on a shared memory system and the consensus about how memory is split in parallel applications is a that shared nothing memory system—as defined by Stonebraker for instance [STO86]—has often proven the better option. The research into GRIP progressed into the work done using parallel Haskell.

Glasgow Parallel Haskell (GpH) [PEY96] is an extension to the pure functional language Haskell [ARG87]. It aims to provide more expressive strata upon which to build sophisticated I/O performance using such techniques such as monads. The goal was to attain implicit semantically transparent parallelism, but the version available uses explicit parallelism [TRI00] by including the *par* instruction in algorithms in a scheme called *evaluation strategies*. Evaluation strategies suggest to the compiler places in an algorithm where parallel processing might be possible; the *par* instruction is used to direct the spawning of new processes.

Evaluation strategies use lazy higher-order functions to separate the specification of an algorithm from its dynamic behaviour (parallelism). The definition of a function has two parts: the algorithm and the strategy [TRI98]. A practical application of evaluation strategies relevant to our subject area is that for accident blackspots. This loads map reference details of accident blackspots from police traffic reports and is typical of a data-intensive complex-query domain. The four phases of the program were tested for parallelism. The results proved to be somewhat disappointing for the authors. The data sets used were small: 1000 accidents occupying .3 Mbytes of store. However, one area that was more promising involved splitting the data on a geographical basis into 'tiles'. This meant sub-sets of the data were safely used in parallel leaving intersection points as the only area requiring special treatment [LOI97].

In our case, there are several areas where the application of parallel processing techniques could be used to improve performance. These are now discussed.

Passing the base expression to all processors

This would rely on domain decomposition being able to take advantage of data placement across all disks participating in the array. The parsing of expressions would be handled simultaneously on each PE that would then be responsible for constructing a query evaluation tree to solve the expression using the data on its own disk. The idea of simply multiplexing an expression across all PEs initially seems a good one. However, there are some drawbacks to this approach.

Queries involving lexical attributes would necessitate inter-process communication to obtain (for example) tokens for strings. This would be difficult to manage effectively and could overload the inter-process bus with data and message passing operations as non-lexical data is transferred between various PEs. The same inter-process communications would be required for each step in a graph traversal operation to ensure no links are missed.

Parallel graph reduction of evaluation tree

Peyton Jones [PEY89] introduces this in a keynote paper. The basic idea is that nodes of a graph are allocated to different processors where there is control maintained over the parent-child reduction sequencing. Difficulties occur when a PE has to fetch or update a non-local node and issues such as object locking and deadlock become relevant. Again, the results are often too fine-grained after considering inter-process communication and the additional costs involved.

Parallel processing at the storage sub-system level

This would include the searching for matching pages that might contain a triple or record. Triple store interface functions such as *ts_present,* used to return a matching triple and discussed in chapter 2 section 2.4.3, could be sent to all PEs thus achieving parallelism. However, the improvement could be small and would need careful co-ordination. Consider the following simple algorithmic analysis of a typical expression (using list comprehensions) to display the last names of all people with a first name of John. Each step of the algorithm is described in words afterwards.

```
      [Lname x||x ← All_emp & Fname name x "John"]

BEGIN query
  1.ts_string_to_token(John) - returns token $T_{John}$
  2.ts_string_to_token(Fname) - returns token $T_{Fname}$
  3.ts_open_set(<$T_{Fname}$,?,$T_{John}$>) - returns $T_{set\_id}$
  4.ts_string_to_token(Lname) - returns token $T_{Lname}$
  5.WHILE there is still a member of the set to retrieve DO
     5.1.ts_fetch_another($T_{set\_id}$) - returns <$T_{Fname}$,$T_{emp}$,$T_{John}$>
     5.2.ts_present(<$T_{Lname}$,$T_{emp}$,?> - returns $T_{curr}$
     5.3.ts_token_to_string($T_{curr}$) - returns last name
     5.4.add last name to print tree
  END WHILE
END query
```

Step 1 involves accessing the triple store to obtain the token for the name "John". T_{John} is the resulting token. Step 2 searches the meta triples for the token for the function name "Fname" and returns the token T_{Fname}. Step 3 passes to the triple store the triple template <T_{Fname}, ?, T_{John}>—where ? represents the entity identifier position. The triple store opens a set of entity identifiers using this template. The returned value T_{set_id} 'points to' this set. Step 4 obtains a token for the other function, Lname, used in the expression and returns a token T_{Lname}. The while loop in step 5 iterates over the entity identifiers in the set accessed through T_{set_id} and in each case: uses *ts_fetch_another* to get a new entity identifier token—T_{emp}, uses *ts_present* to return the token for the Lname—called T_{curr}, before converting the Lname token to a string in step 5.3 and adding it to the print tree in step 5.4.

An analysis of this algorithm reveals the following. Lines (1), (2) and (4) are obvious candidates for parallel processing as they can be considered as separate tasks. However, the allocation of strings across pages is complicated by internal string decomposition—each external string requires splitting into smaller 'chunks' of six characters. Line (3) is dependent on lines (1) and (2) executing correctly and could be done at the same time as line (4). The opening of a set can involve a great deal of searching through many pages—although the set of triples that match the search template are not retrieved as such. Retrieval pages are used to store the identifiers of pages that may contain matching triples. The actual triples are then retrieved as required (in the while loop) under the lazy implementation that is used.

Line (5.1) is clearly a sequential process where the page returned would be held in cache to speed up the next access anyway. Line (5.2) must follow on from (5.1) and, again, must be completed before converting the *Lname* token to its lexeme and adding this to the query tree. Any *token_to_string* function—as with *string_to_token* functions—can incorporate much searching and collating of internal string triples. Larger expressions such as

```
[Lname x‖x ← All_emp & Fname x "John" &
        y ← All_cust & Cus_no y = 12345 &
        Has_ac x = y]
```

are more amenable to parallel execution where the *x* and *y* entity sets can be created on different processors. This is a more promising area to exploit but, at the moment, is again hampered by the structure of the underlying tokenisation. Moreover, the format these expressions take is not fixed: users can create well-formed expressions in several ways.

Domain decomposition

The data domain is centred on triples that are held in a homogeneous triple store repository indexed (primarily) on the first field. For instance data, the first field is most often the relation or function name. Hence clustering tends to place triples with the same function on to the same or consecutive data pages. This makes parallel searching for pages within the current architecture difficult to organise.

Many searches involve strings; and strings are further decomposed into internal string triples for storage purposes. String triples suffer from the same problem as other triples in that they are clustered around their first field—in this case, the first field is used for the string identifier. There are two types of function call that involve searching the domain of string triples—*ts_string_to_token* and *ts_token_to_string*. Both of these involve calls to sub-functions to search for and collate the sub-strings needed that constitute the complete string and might therefore be candidates for parallel implementation.

Internal strings are already searched for in two different ways—either from the beginning of the string or from the end of it, dependent on the length of the string—see chapter 2 and the discussion of the lexical token converter. In order to make the searching of sub-strings amenable to a parallel solution, the sub-strings could be allocated differently (declustered) across pages or disks.

The options for de-clustering of sub-strings are many. They could be spread across processors on a sequential, round robin or hashed basis; they could be clustered on the six characters they hold in their second and third elements. It is possible to allocate blocks of six characters of the search string to different processors. They could even be de-clustered according to their length with different modulo string lengths being allocated to different pages or disks. Each of these schemes would require additional co-ordination and incur the communication overheads that are an inhibitor to parallel processing. Moreover, balanced data placement might be difficult to achieve. In general, it is possible to distribute all data from the homogeneous triple store in—for example—a random distribution method across *n* processors. Because of the difficulties in employing parallel processing techniques within the constraints of the current storage architecture, we suggest a new architecture that tackles the problem at the model level rather than at the physical level.

3.6.3 The vigorous parallelism of AGNA

AGNA [HEY91] is a parallel persistent object system that makes heavy use of list comprehensions and indexes, and pursues parallelism very vigorously at the model level. In the body of a block, all expressions are potentially evaluated in parallel, and the value of the body may be returned as soon as it is available. Also, in primitive applications (including CONS—the list constructor operator) and in function applications where each argument is evaluated in parallel even if not ultimately required. The only exception is for conditional expressions and where data dependency is involved. Optimisations occur in three ways: transformation of comprehensions; translation into dataflow graphs; and translation into code for the multi-threaded abstract machine P-RISC.

The language is also implicitly parallel (the programmer does not specify what must be done in parallel). It uses MIMD architecture where the data (objects) are randomly distributed across the array sequentially. Because AGNA is non-strict, once the first CONS cell has been constructed a reference to it can be returned while the rest of the list is constructed in parallel. Non-strictness also permits the construction of 'open' lists (not NULL terminated), so that the APPEND function can be used to join the lists from each processing element. The results using a uni-processor system showed they were 'within shooting distance' of INGRES. While for multi-processor systems, the optimal processor array size was around eight. Results from AGNA showed it was a fair distance behind relational systems, such as the GAMMA project [DEW90a], but most of it was written in software: there was very little hardware assistance. Moreover, AGNA, as a functional language, enjoys the benefits that the functional paradigm has to offer.

3.7 Introducing redundancy

The use of redundancy is worthy of consideration for any new system nowadays as it reduces the risk of data loss to extremely low levels. Once again, the basic concepts are introduced here therefore readers familiar with these may wish to skip this section. Redundant Arrays of Inexpensive Disks (RAID) [PAT88] link together arrays of smaller, cheaper disks to do the work of larger, more expensive ones. In recent years drive technology has progressed to such an extent that many consider the "I" in RAID now stands for "Independent".

There are six, official RAID levels (0 to 5) proposed by the RAID Advisory Board. But there are other, unofficial RAID levels devised by users for their own particular needs. Recently, the RAID Advisory Board has suggested three new levels to replace some of the confusion that exists. However, for most practical systems the choice is between RAID levels 1, 3 or 5.

RAID controller hardware provides data redundancy to improve reliability. This is done either with a second, mirrored copy of the data disk (as in RAID 1), or by incorporating parity information held on one extra disk that can be used to reconstruct information in the event of data disk failure (as in RAID 3 and RAID 5). This allows RAID systems to continue to operate even if one drive fails. Failure rates are measured in the number of years a disk is expected to work between failures. The Mean Time Between Failure (MTBF) rates are practically insignificant nowadays; using a RAID 5 system with four data disks and one parity disk will give a MTBF of 71,000 years.

Two ways that RAID improves performance are by reducing disk bottlenecks and by increasing disk transfer rates. In the parity schemes, data is allocated to disks on a round-robin basis. It is said to be "striped" across disks in "chunks" of fixed block size. The extra disk needed for parity can either be used entirely for parity information (as in RAID 3) or interleaved with the other disks (as in RAID 5).

The biggest single impediment to RAID is the "write penalty" [FRI96]. The Small Computer Systems Interface (SCSI) method, currently used to interconnect RAID, is likely to give way in the near future to new technology in the form of Serial Storage Architecture (SSA) and Fibre Channel Arbitrated Loop (FCAL). Fibre channels offer 100 megabyte per second data transfer rates and eliminate SCSI bottlenecks. Our ideas for a novel RAID configuration are discussed in chapter 6.

3.8 The physical model for data storage

Shipman—in a keynote paper [SHI81]—crystallised earlier work following the introduction of the functional data model (FDM). Since then, considerable research has been done that embodies some elements of the FDM and applied these to database systems. These include: DAPLEX at CCA, FQL at the

University of Pennsylvania, P/FDM at the University of Aberdeen and, of course, the TriStarp work at Birkbeck College, University of London. Some of these projects were aimed at specific domains—for instance, the P/FDM system was primarily set up for scientific and design databases [GRA92].

Most of the early work on the FDM was done before the relational data model became a sound commercial product. The FDM has stood the test of time because it is well defined being based on very good principles. Because of this, it is evolving into a formalism for the less well defined object-oriented model. Our proposed architecture is, however, motivated by earlier work that contributed to the FDM and very much complements the fundamental principles involved. This is the Associative Data Management System, which is now introduced.

The Associative Data Management System (ADMS) data model [CRO82] uses concepts from set, relation and graph theory and provides the model that we wish to use for our physical storage structure. In ADMS the database is modelled as a directed graph where all the sets of data elements appear as nodes and the directed connections show the relationships between them. The data can be shown pictorially or in tabular form by listing the end nodes on each of the arcs. Additional labelling information is attached to the nodes, which divide the graph into *cliques* of stored record sets. Labels are also used to hold details of access rights for the record sets.

1:1 correspondences and N:1 functions are easily handled by ADMS, but M:N mappings are transformed by introduction of a compound entity set. This replaces the relationship name and means that the arcs are not labelled in any way and thus do not convey meaning between nodes. Nodes are held as "twins" within the database and there is some duplication of sets due to the clustering of cliques into record sets. Two other transforms are done: the introduction of a "dummy" set if a set is related to itself; and a "link" set is created when a group of sets needs connecting at the highest level. This is called the "upper bound".

Querying the database is done by macro substitution using a stack so that plain English words can be used to formulate the user queries. ADMS is used with the CAFS, purpose-built hardware described in section 3.5. Ambiguities can arise when there is more than one path between two sets. In this case the user is asked to select a path. Database update is at the record-set level with a unique data element identifying which records are updateable by which groups of users. Figure 3.10 below shows the ADMS file structure.

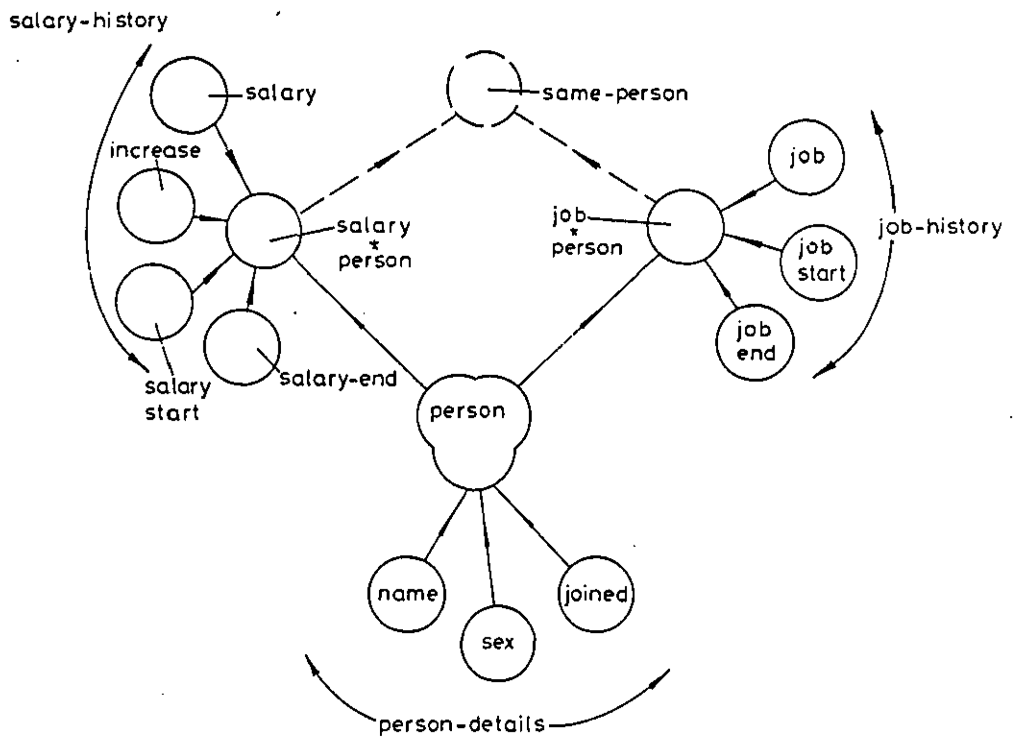


Figure 3.10. The ADMS data model.

3.9 Introduction to our proposals

Our solution to the problems discussed thus far in the thesis is that a new architecture is required that incorporates: parallel processing, improved string manipulation and other areas of general functionality, and redundancy.

3.9.1 The data model

Some data (for instance meta data) does not form part of the function graph model and could be stored differently. Attribute data does not form such an important part in graph traversal, often being used only at the start and end of a series of traversal operations. This leaves just the entity data—the links between entities—as 'real' triples that are used in the graph traversal process.

So, the underlying structure for our architecture involves storing entity-to-entity triples separately from entity-to-attribute records. Included in this is the storing of attribute-to-token mappings (string tables being the most common example) separately from the entity triples and attribute records. Our architecture is discussed in chapter 6.

3.9.2 Improving string handling

Once strings are more loosely structured, faster searching techniques, such as those found in [BOY77, HOR80] can be implemented. Moreover, it would be desirable to add functions to perform stem matching (truncation)—left-hand end searching, right-hand end searching (or both), and enable the use of wild card characters for elastic matching. A split of the generic type *string* into two sub-types representing short attributes and text, would allow for more control over searching operations generally. Our proposals for string handling and extending search options are discussed in chapter 4.

3.9.3 Extending interface functionality

One of the key elements in the original proposals was the adoption of a simple set of interface functions; the benefits of this approach were identified in chapter 2. However, we believe interface functionality can be extended for the storage level interface without compromising the model level. These extensions are outlined in chapter 5. Additionally, functionality can be enhanced at the model level for strings (in particular) and in other more general areas incorporated in optimisations. These are discussed in chapter 7.

3.9.4 Making the architecture parallel

These proposals include ensuring that the chosen architecture is amenable to parallel processing techniques. There are several ways this can be done and some of them have been covered earlier in this chapter. Our choice is to adopt a MIMD taxonomy with dataflow. The design of this and the decisions taken are set out in chapters 6 and 7. The creation, population and maintenance of a database using our architecture are discussed in chapter 8. Here we use a North Yorkshire Police crime database used with test data based on real-life crimes.

3.10 Summary

Following chapter 2, where we described the strengths and weaknesses of the current implementation base on a triple store architecture, this chapter has introduced the background to other areas of work that form part of our proposals. These include alternative grid file implementations and other storage and access methods, the fundamentals of binary relational storage structures and related data models. Additionally, parallel processing and redundancy are described, as is the basics of text searching and the importance of user requirements. Finally the proposals for the areas of work discussed in later chapters are set out.

Chapter 4 Enhancing string manipulation

4.1 Introduction

We highlighted in our introduction that string handling was one of the weak areas identified in the trials of the TriStarp system [KIN96a]. In this section we discuss string handling, dividing our attention between two strategies where improvements can be made. These approaches are:

- enhancements to the current implementation and
- identifying possible improvements using the new architecture.

We begin by showing how changes to the current system can easily achieve large performance gains. This is done within the framework of the triple store architecture for all data. We then describe how de-tokenising strings can lead to improved searching techniques. Moreover, by using a new data type for large, document-type strings, additional functionality for string manipulation becomes possible. Results are given for improvements achieved within the current architecture and those for a new architecture. We summarise these in the context of other functional data languages and object systems and show how object-oriented concepts—underpinned by the functional model—are being incorporated into the relational model [MEL02].

4.2 Enhancements to the current software

The current built-in, object-level, string-manipulating functions, their descriptions and examples of their use, are shown below. (Note the terms 'object-level' and 'built-in' are used synonymously through this chapter.)

function	description	example
concat	takes two strings, joins them and returns the resulting new string	concat "abc" "def" returns string "abcdef"
length	takes a string and returns its length	length "abcdef" returns integer 6
substr	takes a string and two integer parameters and returns a sub-string from the position of the first integer to the position of the second integer	substr "abcdef" 2 5 returns string "bcde"

Table 4.1. Current built-in string functions.

Any string manipulation is accomplished by the creation of user-defined functions that can include the above object-level functions as part of their definition.

Queries are parsed in the following way—see [POU89] for a detailed description. A query tree is constructed which breaks down an expression into a collection of nodes to be evaluated by the compiler. The pattern-matching algorithm evaluates the query by recursively reducing the nodes of the query tree by a process known as eager graph reduction. This involves evaluating the children of node *n*, then evaluate node *n* itself, finally replacing *n* by the result. The evaluation continues until the root is reached and a result can be returned to the user.

There are standard arithmetic object-level functions `+`, `-`, `<`, `>`, `=`, `*`, `/` (integer division) and `%` (the modulo function) for use in queries, plus the three string manipulating functions—*concat, length* and *substr* mentioned above. Other functions for use in queries are termed user-defined and have to be coded directly at the model level or loaded as part of the environment at database creation.

As a query is evaluated, object-level functions may be encountered. In this case, the compiler breaks off the graph reduction process and evaluates the called function with the given parameters. The current node of the query tree is then replaced with the returned value of the called function and the evaluation continues from there.

For simplicity, strings are held in the query tree in tokenised form, along with names of functions etc. This enables a query tree to hold nodes of fixed length and aids compilation as all comparisons are made between tokens. However, when an object-level, string-manipulating function is called, any strings have to be re-constructed into their full forms before the object-level function can be invoked.

As an example, consider the user-defined function *contains* which searches for a pattern in a text and returns a Boolean result. The function *contains* is itself defined in terms of another user-defined function *search* as follows `contains(text,pat) <= search(text,pat,1).` The function declaration and definition for *contains* and *search* are given below where prefix notation is used.

```
search : string string integer -> bool /* function declaration */

search text pat n  <=                  /* function definition */
      let a == length pat in
      let b == length text in
            if > a (+ - b n 1) false
            /* else */
            if = pat substr text n (- + n a 1) true
            /* else */
            search text pat (+ n 1)

contains : string string -> bool  /* function declaration */

contains text pat <= search text pat 1
```

A use of this function might be `contains "abcdef" "def"` which would be evaluated as follows.

1. The whole expression is parsed to form a query tree that involves crossing the level 0 interface to generate and return tokens for `"abcdef"` and `"def"`. The evaluation process begins by calling `contains` and replacing it with `search` and its parameters. Now included is the integer 1 for the string starting position.
2. `search "abcdef" "def" 1` is called. The lengths of the two strings are calculated using object-level function *length* and stored as local variables adding branches to the query tree.
3. A test is made to see if the search pattern is longer than the remaining portion of the text. If it is, the function exits returning false. Otherwise, a comparison has to be made between the search pattern and a part of the text of a similar length. The object-level function *substr* is then called.
4. `substr "abcdef" 1 3` returns `"abc"` (the 1st to 3rd characters of the string). This involves building more branches of the query tree and crossing the level 0 interface to generate and return a token for the sub-string `"abc"`. If the string tokens are the same the function exits returning true. If not, a recursive call is made to *search*.
5. `search "abcdef" "def" 2` is called and the process continues from 2 above until the token for text string `"def"` matches the token for the pattern `"def"` at which point true is returned to the user.

The initial call to *contains* results in one call to function *search*, which then calls itself three times before the answer true is returned. During the process five string tokens are created—two for the initial parameters `"abcdef"` and `"def"`—plus three others for the intermediate patterns `"abc"`, `"bcd"` and `"cde"`. Only the first two tokens are meaningful in this case.

The recursive nature of these expressions, together with tokenisation and the continual crossing of the level 0 interface, is what leads to the unsatisfactory run time for such evaluations. Moreover, in large searches, the erroneous tokens created quickly deplete the token space available for strings—sometimes to the point of exhaustion, thus rendering the software unreliable and compromising data integrity. Object-level functions only add tokens to the database for complete strings and only then if they do not already exist.

We have extended the concept of string handling, object-level functions from the above three primitives to include a much wider selection of functions that work in the same way. These can access the storage sub-system more quickly in the graph reduction process so execution is faster. Although the text and search strings have to be re-constructed before the function is called, this strategy obviates the laborious tokenisation and comparison of sub-strings in the text to be searched. The compiler can now handle these searches in just one non-recursive function call.

We first provided 15 experimental, object-level, string matching functions from which other, more complex, string matching operations can be constructed by the user. The set includes features found in standard text retrieval systems and is sufficient for the majority of user needs. The meta characters used in our functions are as follows

"_"	match any one character
"%"	match zero or more characters
"|"	OR operator
"&"	AND operator
"<"	must-come-before operator

The choice of meta characters is arbitrary, although the elastic matching character "%" and match any one character "_" are taken from the syntax of SQL, any other symbol could be used. We have tried to keep the meta characters intuitive though. The functions provided, their description, and examples of their use are given in Table 4.2 below.

name	description	example
em1	Looks for exact match words using a move one-at-a-time strategy.	em1 "here it is" "it" – true em1 "here it is" "and" – false.
em11	Like em1 but uses a shift table.	As for em1.
em2	Like em1 but allows for "_" character in pattern.	em2 "here for" "h_re" – true. em2 "here for" "r__e" – false.
em21	Like em2 but uses a shift table.	As for em2.
em3	Allows for right-hand truncation in the search pattern via "%".	em3 "aping" "ap%" – true. em3 "aping" "app%" – false.
em4	Allows for left-hand truncation in the search pattern via "%".	em4 "ended" "%ded" – true. em4 "ended" "%ds" – false.
em41	Like em4 but uses a shift table.	As for em4.
em5	Allows elastic matching character "%" embedded in search pattern.	em5 "right" "r%ht" – true. em5 "right" "r%gh" – false.
em6	Allows "%" character at either end of one-at-a-time search pattern.	em6 "right" "%igh%" – true. em6 "right" "%ug%" – false.
em61	Like em6 but uses a shift table.	As for em6.
mm1	Pattern contains multiple search terms separated by "|" characters.	mm1 "jo vic" "vi|vic" – true. mm1 "jo vic" "vi|val" – false.
mm2	Pattern contains multiple search terms separated by "&" characters.	mm2 "jo vic" "vic&jo" – true. mm2 "jo vi" "vi&al" – false.
mm3	Pattern contains multiple search terms separated by "<" characters.	mm3 "jo vi al" "jo<al" – true. mm3 "jo vi" "vi<jo" – false.
ss1	Uses on-at-a-time search strategy to return occurrences of pattern in text.	ss1 "one to one" "one" – 2. ss1 "one two" "too" – 0.
ss2	Like ss1 but used shift table.	As for ss1.

Table 4.2. Experimental string functions.

All of the above functions have the type signature: *string string* → *bool* except *ss1* and *ss2* which have type signature: *string string* → *integer.* A smaller set of functions was then written that (generally) return the word that forced the match. This is more significant for searches that involve elastic matching or conjunctive search patterns. These functions are shown in Table 4.3.

name	description	example
matches	Returns first word that matches *pat* or the empty string	matches "here and now" "%nd" returns "and"
rest	Finds first word matching *pat* and then returns remaining text or the empty string.	rest "this and that or" "and" returns "that or". rest "this and that" "that" returns "" (the empty string).
or_str	Allows the '|' character as OR function. Returns first word matching *pat* and returns it.	or_str "one and two" "six|two" returns "two". or_str "one and two" "nine|ten" returns "" (empty string).
and_str	Allows the '&' character as AND function. Returns boolean.	As for mm2 above.

Table 4.3. Object-level string functions.

If used in list comprehension as part of a filter, functions that return strings have to be embedded in an expression that returns a Boolean result. For example, to list the *ref_num* of all *persons* where attribute *name* is like "Fre%", the following expression is needed.

```
[ref_no x‖x ← All_person & not = "" matches name x "Fre%"];
```

The filter might look confusing because of the prefix notation, but it is saying 'only display the *ref_no* for persons where the search for name beginning "Fre..." does not result in an empty string being returned'. It is perhaps more natural to use these functions in user-level functions such as in the following example where we are trying to find all words like "WINDOW" in a crime database described next. Function *f* below again uses prefix notation.

```
f : string (list string) -> (list string);      /* declaration */

f a [] <= [];                                    /* definitions */
f a [h|t] <=
      let x == matches h a in
            if = "" x f a t
            /* else */
            [x | f a t];

f "%IND%" map (scp) All_crm;                    /* usage */
```

The function *All_crm* is a zero-argument generator function of which there are equivalent functions for all non-lexical types. *All_t* returns the current extent of type *t* as a set. The *map* function is a second-order function that (in this case) maps the function *scp* over *All_crm* to produce a list of crime reports that is itself passed to function *f* as described above.

The function *scp* represents the "scene of crime report" attribute from the entity *crm* in the crime database. The crime database is used in an operational environment by the North Yorkshire Police Force and holds test data based on 2,500 reported crimes. There are 948,000 triples held in the triple store—where approximately 314,000 hold strings for scene of crime reports. Appendix A2 gives a fuller description of the triple breakdown. A small part of the schema is reproduced below—the schema is shown in full in appendix A1.

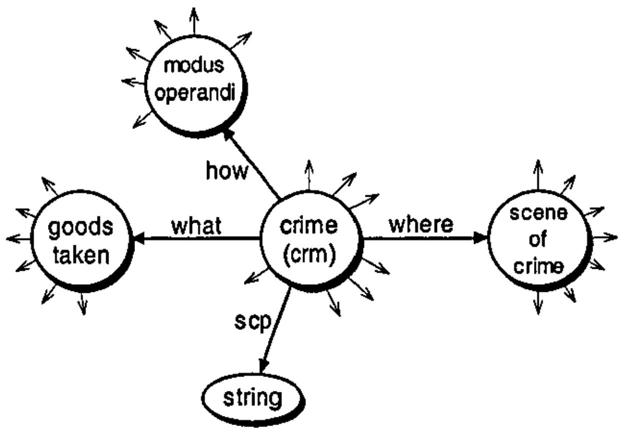


Figure 4.1. Crime database schema (part of).

For the remainder of this section, we will use examples from Table 4.2 as these return Boolean results and are thus easier to read in list comprehensions. A comparison was made between our exact-match, object-level function *em1* and the user-defined function *contains* against the scene of crime data held in the crime database.

4.2.1 Comparisons between object-level and user-defined functions

The results of comparisons are first shown graphically in Figure 4.2 and then in tabular form. Table 4.4 below gives the execution times in seconds of four runs of an expression that uses the two types of functions in list comprehensions. Each expression uses a global variable ($c) to hold a sub-set of the entity identifiers for crimes. This sub-set is allocated with the user-defined function *take*—thus `$c == take 1 n All_crm`. All runs were made on a Sun SPARC station 2

```
count [scp x ‖ x ← $c & contains scp x "DOOR"];        (1)
count [scp x ‖ x ← $c & em1 scp x "DOOR"];             (2)
```

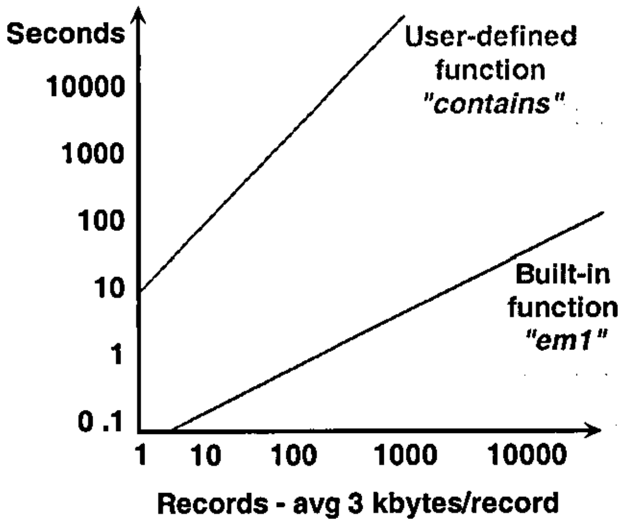


Figure 4.2. Improvements in string functions.

$c	times for (1)				times for (2)			
1	9	9	9	9	< 1	< 1	< 1	< 1
10	139	142	137	139	< 1	< 1	< 1	< 1
100	1486	1457	1443	1459	1	< 1	1	1
500	10244	9997	10053	10330	3	2	3	3
1000	estimated at > 19 hours				6	6	6	6
2500	estimated at > 49 hours				18	19	18	18

Table 4.4. Comparisons between *contains* and *em1*.

Not surprisingly, there are substantial time savings using *em1* instead of *contains*. This is clear from the above table and can be accounted for by the fact that *contains* has to cross the level 0 interface so frequently, whereas *em1* does not. The expression that uses *em1* removes the majority of the *ltc_str_to_id* operations—some of which also generate new triples. In fact, using *em1* creates only one extra triple—the one for "DOOR". In contrast, using *contains* means the creation of 47,163 extra triples—all of which are needed only for the comparison and are otherwise useless.

The timings of the four runs were close which suggests there is little to choose between asking the LTC to generate a new token and then pass it back, or asking the LTC simply to pass back an existing token. Another advantage is that using object-level functions like *em1* does not deplete the token space available for string tokens. The built-in search functions described in this section are shown in appendices A7 and A8. They were incorporated into a new system we called FDLS (FDL Strings). The searching techniques are described in section 4.2.3.

4.2.2 Comparisons between conjunctive and disjunctive search types

Leaving aside the clear superiority of object-level functions we now wish to compare two ways of handling conjunctive and disjunctive searches. We have provided these options as multi-match, object-level functions *mm1* and *mm2*. The same results can be obtained with user-defined functions *and* and *or* used together with our exact match function *em1*. Consider the following expressions.

```
count [scp x ‖ x ← All_crm &                                  (3)
           (em1 scp x "DOOR") or (em1 scp x "ROOF")]

count [scp x ‖ x ← All_crm & mm1 scp x "DOOR|ROOF"]           (4)

count [scp x ‖ x ← All_crm &                                  (5)
           (em1 scp x "DOOR") and (em1 scp x "ROOF")]

count [scp x ‖ x ← All_crm & mm2 scp x "DOOR&ROOF"]           (6)
```

We note that tokens would be created for the patterns "DOOR|ROOF" and "DOOR&ROOF" but these are the only ones that have to be added. We used the above four expressions against all the 2,500 crime reports (average length 750 characters). The number of search terms used was 2, 10 and 50. These numbers were chosen to show a trend in increased search times and can be thought of as *x*, *5x* and *25x* accordingly. There were several runs of each test which was done on a Sun SPARC station 2. The results are shown in Table 4.5 below with times in seconds.

search terms	expression (3)	expression (4)	expression (5)	expression (6)
2	16	14	14	14
10	17	15	17	14
50	25	15	22	15

Table 4.5. Timings for different types of *or* and *and* functions.

From this table we can show, in Figure 4.3, that our multi-match functions scale to provide an increasing advantage over using the standard *or* and *and* functions.

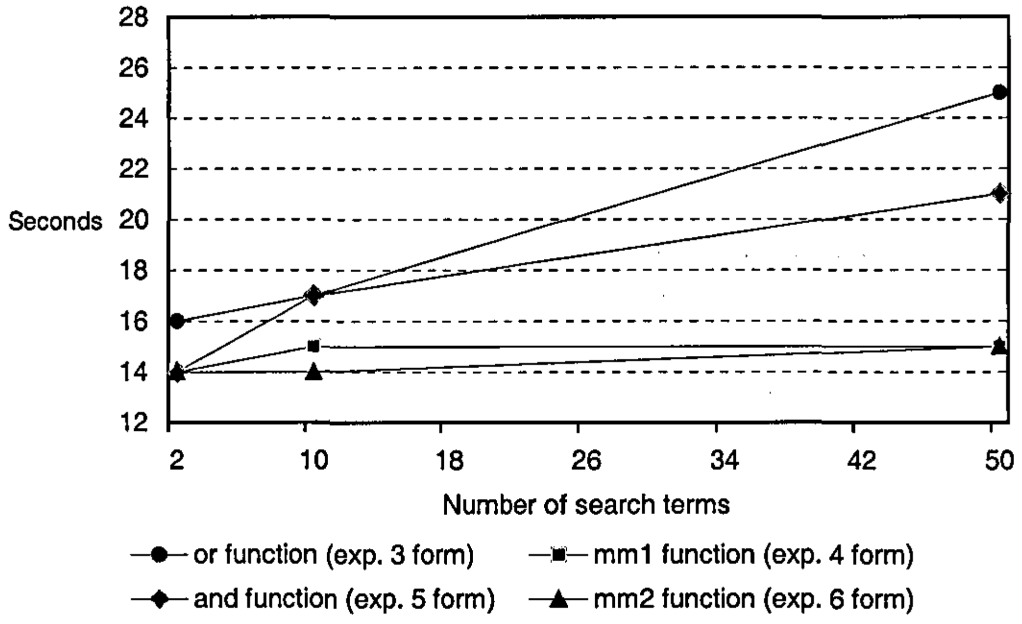


Figure 4.3. Comparisons between different *or* and *and* functions.

A closer inspection of these figures reveals the expected result of *and* operations executing faster than *or* operations. There is little to choose between either class of function when used against a small number of search terms. However, as the number of search terms increases, the advantages of the multi-match functions become clear. This is because the standard *or* and *and* functions increase the size of the query tree. We also experimented using a local variable to hold *scp* x in (3) and (5) via a **let** ... **in** construct, but this made no difference to the timings. A feature not investigated was filter optimisation; both the multi-match functions search for "DOOR" before searching for "ROOF". Further runs showed that the relationship between numbers of records searched and execution time is roughly linear.

4.2.3 String searching techniques used

The object-level functions use two forms of searching techniques. Functions such as *em1* match the pattern against the text from the right-hand end of the pattern working left until the pattern is found or a mismatch occurs. Following a mismatch, the pattern is moved along the text by one character and the search process starts from the right-hand end again.

The other form of search, which is used in functions like *em11*, uses a shift table to calculate the amount the pattern can be moved along the text when a mismatch occurs. This is based on the Boyer-Moore-Horspool (BMH) algorithm [HOR80] that demonstrates a practical application of the generic Boyer-Moore (BM) algorithm [BOY77]. The superiority of the BMH algorithm over the BM algorithm is achieved by simplifying the pre-processing of the text and search pattern. Only one of the shift tables suggested in [BOY77] is used—*delta1*—so pre-processing is kept to a minimum.

Our shift table is based on the BM *delta1* table and is sufficient to allow the pattern to be moved along the text more rapidly so that, on average, two thirds less comparisons are necessary. The *delta1* table sets up a shift array for each letter in the alphabet, where an integer indicates by how much the pattern can be shifted to the right. If a letter is not in the search pattern, its integer is equal to the length of the search pattern because we can safely move the search pattern past the mis-matched letter without fear of missing any potential matches.

Because we use a *delta1* table only, time is saved setting up the relatively under-used *delta2* table proposed in [BOY77] and demonstrated in [HOR80]. The *delta2* table searches for discovered sub-patterns in the search pattern and uses this knowledge to check if a greater move for the search pattern can be made than the *delta1* table alone would suggest.

For patterns of length five or greater, the BMH algorithm is the better option [HOR80]. However, we used our functions *em1* and *em11* on identical queries involving multiple search terms over the *scp* function in the crime database (average crime report length 750 characters). There was no difference. When we consider the functions that allow for missing characters and elastic matching, there is even less to be gained by pre-processing the text because the "_" character in the pattern can be matched against any character in the text. This means the pattern will move more slowly through the text anyway. The conclusion being that, in this case, pre-processing the strings is not worthwhile.

4.3 A different approach to string handling

The previous section showed that substantial improvements are possible by changes in functionality to the triple store architecture underpinning the current software. If the architecture for string storage were changed it may be possible to remove the weaknesses of tokenising strings while retaining the advantages that tokenisation offers.

In this section we endeavour to show that a new architecture for strings can accommodate the benefits of tokenisation with a more flexible method of string storage. Moreover, this can be done in such a way that powerful searching facilities can now be made available to the user that have hitherto proved difficult to provide.

4.3.1 A new type for large text documents

Part of our proposals include the introduction of a new type we have called *text* to be used for large, document-type attributes, where text can be considered a list of strings. A synonym for string could be word, so text could conveniently be considered as a list of words. This would permit more powerful text searching facilities to be provided—ones that would not necessarily be applicable to shorter attributes.

Shorter attributes typically contain between one and a small number of words that form a single semantic unit. Words included in a text, on the other hand, have a looser connection within the context of the whole document. Each word in a text document is delimited—often with the space character, although this need not be the case. A decision would have to be made upon database creation and then used as a semantic rule to guide users when entering queries. More details of the structure of words in texts will be given later in this chapter.

For the rest of this section we shall refer to the short string types as type *string* and the longer string types used in documents as type *text*. The dividing line between whether an attribute should be a string type or a text type is application dependent and would be set up at the model level as part of the database schema. By making a design decision like this, it is possible to devise string-searching functions that discriminate between the two types. Any improper use of these functions would be flagged as type errors in the usual way by the type checker.

We note that there is plenty of scope for further sub-typing of the generic type *string* and our new type *text*. The concept could be extended to cover html documents for example. This is discussed later in this chapter.

4.3.2 Combining strings and tokens

The reasons that strings were tokenised before being stored in the triple store are as follows:

1. duplication of strings is removed
2. compact representation of strings
3. compact storage into pages on disk and main memory
4. uniformity of fixed-length tokens for the compiler
5. adherence to the philosophy of homogeneous triples
6. to maintain simple interface functionality.

To allow for searches to contain missing characters, cover case and word contraction inconsistencies, allow for left- and right-hand truncation and embedded elastic matching characters in a search pattern, it is desirable to hold strings in their full text format somewhere in the storage sub-system. There are several options available. Leaving aside text attributes for the moment, string attributes have different possible representations:

1. the complete multi-string attribute could be tokenised as is the case now
2. each word could be tokenised with these tokens held in sets
3. the actual words themselves could be held in full wherever they appear in the database
4. a combination of the above.

An example mapping of 1 might be "white European" $\rightarrow$ 3278242123. An example mapping of 2 might be "white European" $\rightarrow$ {1536691292, 3231125932}. In each of these cases, the string "white European" would itself be broken down into sub-strings for storage in the triple store. We believe strings should be stored in full format in a string table used to provide the mapping *string* $\rightarrow$ *token*. This is discussed in the next section. However, the alternatives to be considered are as follows.

The first idea for the triples was that one token would represent each unique string (as in 1 above) and we proposed to store the attributes in sets with the following structure:

$$<\$e, (\$r_1, \$a_1), (\$r_2, \$a_2), \ldots , (\$r_n, \$a_n)>$$

Where \$e refers to an entity surrogate; \$r refers to a relation surrogate and; \$a refers to an attribute token. Each field is of fixed length. This situation would synthesise the benefits of data compression on disk and in memory, and provide the more direct access to strings that is required via the string table.

However, we believe it is desirable to be able to identify individual words within string attributes. Therefore, we propose breaking the attribute down into delimited words before storage in the string table. Each delimited word is held in alphanumeric order (case folded) and maps to a unique token. This means that more powerful searching operations are available to the user. Firstly, searches for words can be ordered—e.g. "find attributes where the word "Venetian" comes before the word 'blind'". Secondly, quorum operations can be used—e.g. "find an attribute with at least two of these words "burglary, arson, violence, firearms, aggressive, mugging" somewhere in it. Moreover, left- and right-hand truncation are more easily catered for.

String searching functions would then proceed by searching for a string in the string table and then return a set of tokens and an indication of the number of times a string occurs in the database. The results of these functions would be passed to the parser and used lazily in any remaining sub-expressions still to be evaluated in the list comprehension. The choice of what should be stored in the attributes is then between holding the full text representation, or holding sets of word tokens. These options are now considered.

Holding sets of word tokens

This would require that each word (stop words excepted) be mapped to a unique token and that each entity attribute would then have the following kind of format:

$$<\$e, (\$r_1, \$a_{11}, \$a_{12}, \$a_{13}), \dots , (\$r_n, \dots)>$$

Where $\$a_{11}$, $\$a_{12}$ and $\$a_{13}$ each refer to a separate word in the string attribute. As before, each field is of fixed length.

This option would combine the compactness of tokenisation with the improved level of granularity for string storage. However if a set of tokens is stored for each string attribute there is unlikely to be much saving of space and, any savings there were to be made would be negated by the necessary reverse mapping *token* → *string* that would be needed for printing and display purposes.

The tokenisation of individual words could be done in a compressed format dependent on the $\log_2$ of the number of words held in the string table. This method has been used in the Hibase project [COC98] and requires variable length tokens. It seems intuitive to maintain word token allocation from the sub-domain of 32 bits that are set aside for strings so there is uniformity with the tokens of other types. It is also easier for data filters to scan fixed-length fields; variable length tokens would require an identifier byte to indicate the token length and would negate the benefits of using tokens in the first place.

Holding actual strings in the attributes

An alternative would be to hold the actual strings in the attributes and remove the *token* $\rightarrow$ *string* mapping completely—although the tokens would still be used for the inverse function mapping. The mapping of very long strings to tokens of only 32 bits had the advantages mentioned earlier. However, as we now want to hold strings at word level so we can do more with them, the option of holding full format strings in the attributes does not carry such an overhead. If full format strings were held the following would hold:

1. there would be duplication of data—as indeed there is with tokens in the current system
2. self-identifying formats would need to be added, e.g. word lengths. These would take the place of the "space" character now no longer required
3. the average English word is six or seven characters long – equal to 48 or 56 bits required. This is an increase of 50 to 75 percent on the 32 bits currently needed for storage
4. there would be no need for a *token* $\rightarrow$ *string* mapping for display purposes.

In support of the argument for holding full strings in attributes, a comparison between the new proposals and the criteria for tokenisation, should be made. So, to rationalise these proposals with points 1 to 6 at the start of this section the following are noted:

(i) there would be duplication of strings in the attributes; this is a situation that has existed for many years with, for instance, Oracle databases. With the continuing reduction of disk costs, it is not as much of an overhead today as it once was. Moreover, space is not one of the main design issues (refer to our summary of design considerations in chapter 3)

(ii) tokens can represent large strings compactly but the strings still require decomposing into internal sub-strings for storage in the triple store. Furthermore, our plans for text attributes will involve a text identifier being generated and held in the record with the full text document held elsewhere. The text identifier would therefore act in the same way as a token identifier does in the current system. This is discussed further in a later section

(iii) compact storage on data pages would not be so easy to achieve with variable length fields. However, as the types of database for which this architecture is being developed are likely to be very large, to be of a textual nature, and have fairly static data, then data volatility and its inherent problems are less applicable. The records would be placed compactly at database load/re-organise time, with subsequent records added to a temporary area pending database re-organisation.

Compact storage in memory would no longer be a prerequisite as the use of a data filter would obviate holding large numbers of pages in memory many of which are not accessed anyway.

(iv) the benefits for a compiler using only fixed-length tokens is perhaps a little tenuous. Compilers tokenise their own expressions anyway, so this problem should be easy to solve. As was shown earlier in this chapter, removing unnecessary tokenising of data and devolving functionality to a lower level in the execution process allows for a substantial improvement in performance

(v) some data is clearly more suited to the triple concept—entity-to-entity triples and meta data for example—and these we intend to keep tokenised and held as before

(vi) interface functionality needs enhancing so that data storage reflects data usage in a way that boosts performance. This can be done in such a way that does not compromise the robustness of the interface protocols that are currently in place. This is discussed further in chapter 5.

In summary, there is no need to maintain a *token* $\rightarrow$ *string* mapping for displaying the records if full text strings are held in the attributes. A string table provides the *string* $\rightarrow$ *string_token* mapping that will enable token sets to be collated for further retrieval. They also provide the function mapping *string_token* $\rightarrow$ *entity_id* that is frequently used in query expressions. Lastly, there are now very precise things that can be done in text searching and the type *text* allows for specific functionality to be targeted to larger documents. In the next section we show how the data structures will synthesise the storage of strings and tokens to allow for the mappings that we wish to provide.

4.4 The data structures

In this section examples of the data structures and mappings are given. The data structures required are as follows:

- the string tables that provide the string → string_token mapping
- the string triples that provide the string_token → entity mapping
- the other lexemes—integer, real, etc
- the attribute records, and
- the text type attributes—now referred to as documents.

4.4.1 The string tables

Each word used in the database is held in a string table. A word is defined to be a delimited sequence of alphanumeric characters that has a meaning on its own. The delimiter would be application specific but would often be the space character. The string table is held in alphabetical order (case folded) on disk and accessed by a coarse indexing structure (B-tree). This will enable string searches to move rapidly to the relevant sub-section of the table that is then scanned by the filtration hardware. The index could be ordered on (say) the 26 letters of the alphabet, although this again would be application specific. The structure of each entry in the string table is shown in Table 4.6 below.

string	string_token	length	occurrences
full ASCII representation of a string	fixed-length from sub-domain for strings	integer for the length of a string	integer shows how many times word occurs in relations

Table 4.6. String table record structure.

The string token identifier is used for the collation of string token sets and also in the inverse function mapping *string_token* → *entity_id*. A count of the number of occurrences provides a "fast track" into the system. It is used to enable users to decide whether they wish to abort a query if, for instance, there are too many 'hits'. This lets them refine their search. An example of this structure is given in Table 4.7 below where #n indicates the use of an arbitrarily generated string token from the sub-domain of 32 bits used for string tokens.

string	string_token	length	occurrences
analyst	#49	7	2,195
programmer	#3741	10	74,545

Table 4.7. Example of string table.

The string table is then duplicated where the second copy holds the strings in reverse order—e.g. "Fred" changes to "derF"—the other three fields being the same. This enables searches from either end of the string to be made. Although the mapping *string_token* → *string* is not required under our scheme, there is a simple way this reverse mapping can be handled if it is necessary. This involves holding a B-tree index on both the string and the string token.

4.4.2 The string triples

The unique token allocated for each string is used in the string triples data structure to provide the mapping between strings and the entities to which they are related. From the example data in Table 4.7, the string triples might be as shown in Table 4.8 below, where $ indicates an entity surrogate taken from the sub-domain of 32 bits reserved for entity identifiers. The relations are shown in full but would in reality also be tokens.

relation	string_token	entity
has_skills_as	#49	$343218
has_skills_as	#3741	$123456
has_skills_as	#3741	$128332
job_title	#49	$923432
job_title	#3741	$732119

Table 4.8. String triples.

Each of the three fields is fixed length so these triples are held compactly on disk indexed on the first field, within that on the second field and within that, on the third field. Although this structure is similar to the triple store concept, it need not be indexed as such. It is sufficient to use a B-tree structure indexed on ascending relation order. As new entries are added to the string tables, string triples are inserted to give the mapping to entity identifier. There is no correspondence between the alphabetical ordering of strings and the ascending order of string surrogates used. Therefore, the relation name is the primary index key for string triples.

4.4.3 Other lexical types

Other lexical or base types include *integer, Boolean* and *real.* The mappings for these types are trivial in comparison to string types. Integers merely drop the top three bits that are required for the type signature leaving 29 bits for the value. A similar arrangement for real type adheres to the IEEE 754 binary standard for double precision floating point numbers. Boolean types have three values: true, false and maybe, where reserved integers are used as tokens for each. The lexical triples for these types simply hold three fixed length fields comprising the relation, the lexical value and the entity identifier. These triples are ordered on the first field (relation). *Text* and *BLOB* types have an identifier in the second field that is again allocated from a discrete domain.

4.4.4 The attribute records

Storing the actual string in the record would give a structure like:

$$\langle \$e, \{\$r_1, \ldots, \$r_n\}, \{A_1, \ldots, A_n\}, \text{timestamp} \rangle$$

Where \$e refers to an entity identifier, \$r refers to a relation surrogate (and its offset) and A_n refers to an attribute. The words within each attribute would be based upon an agreed delimiter—often the space character. A record structure makes it easier to display all the direct attributes for a particular entity without the need to perform a *token* → *lexeme* mapping. Because of the string triples, the string attributes of a record are not called upon to provide the inverse mapping that is often needed in expressions like:

```
[{age x, fname x} || x ← Inv_lname "Smith"];
```

In this expression the string tables and triples would be used to obtain the people with the last name "Smith". Then the attribute records for these people provide, directly, the age and first name without a *token* → *lexeme* mapping being required.

4.4.5 The structure of text documents

In section 4.3.1, we introduced a new data type *text* that is used to hold a set of strings that has a looser connection than the shorter string attributes. The strings that make up a document of type *text* will also need delimiting to word level, so the structure of a document needs to be able to cope with this. All the delimited words in a document are included in the string table and are mapped to a unique string token for that word. The token is then used in the string triples to identify which entities contain that word and what relations join the two. The records themselves will not hold any text. The value for an attribute of type *text* is a document identifier that indicates where the text document is held. Figure 4.4 shows part of a record that has two text attributes in its structure.

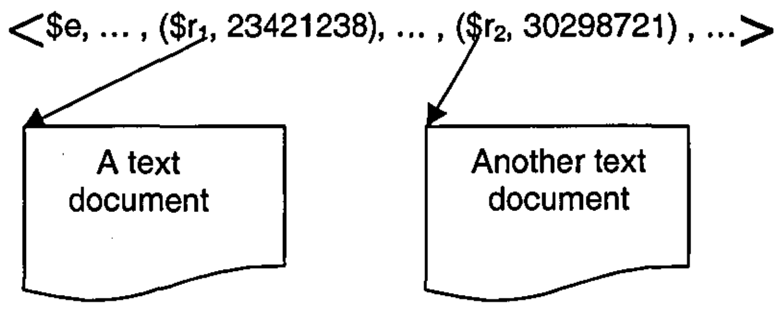


Figure 4.4. Document mapping.

The documents are held as 'link' files and contain the full text with various identifiers included. These are now discussed.

Documents usually have structure with headings, abstracts, titles, sentences, numbered paragraphs etc. The more the software knows about a document the more helpful it can be in enabling users to refine searches. There has been an historical progression in the formatting standards for documents, culminating in the XML/XHTML standards—discussed later. However, for our purposes we refer to the Textmaster document system [KAY85] which uses a normalised document format (NDF), based on the Office Document Architecture (ODA) [KOC94, FAN92]. This allows documents to be structured in three ways:

1. the structure of the document is internally defined using the built-in features of NDF
2. the structure of the document is not defined using NDF but instead uses markers edited into the text content. This allows documents to be prepared on equipment that offers no support for NDF

3. the document is regarded as being unstructured text. However, for filing and retrieval purposes a number of document attributes can be defined—author, title, etc.—that are then held in a separate document card file (called a document profile in ODA).

The first two cases are more suitable for highly structured documents; the third case is more appropriate to more loosely structured documents such as memos, informal minutes and possibly crime reports. In this case the structure will tend to vary from one document to another and the organisation imposes very little control over the way they are written or typed. Often the person filing the document has responsibility to create some order out of the text but has no authority to alter it in any way. The document card file would carry the structure of each document class—where document classes could be: memo, minutes, crime report, etc. The entry of a document would then adhere to the constraints that apply for the document class.

The setting up of the allowable attributes and fields for each document class would be a centralised task so that, once done, a uniformity of document entry can be maintained for all users. It's impossible to decide generally what fields would apply but there may be a heading field, some key words and perhaps even the word delimiter could be document specific. Document layout can also be defined so that displaying a document will always be done consistently.

The Textmaster system keeps two copies of each document—one for display or printing purposes that holds formatting information etc., and another that holds the normalised text in self-identifying format for rapid searching. (Here, normalised text is text which has had the stop words removed from it.) We consider that the formatting information could be kept in the document file card as part of the meta data, while leaving the actual document with just the field identifiers and word lengths in it. At this stage it is not proposed to remove the stop words from the text document, although they would not be recorded in the string tables.

Scoping identifiers, field codes and trailer records are used to delimit sequences of words, sections etc., and are added at appropriate points in the document. These identifiers enable quorum searches—and, indeed, searches in general—to be restricted to sections, paragraphs, etc. depending upon the structure of the document class. Proximity searches can also be made between terms that appear in: (1) the same sentence, (2) the same paragraph, (3) the same field, etc. Although the field types are very application specific, Figure 4.5 gives an idea of how this is done.

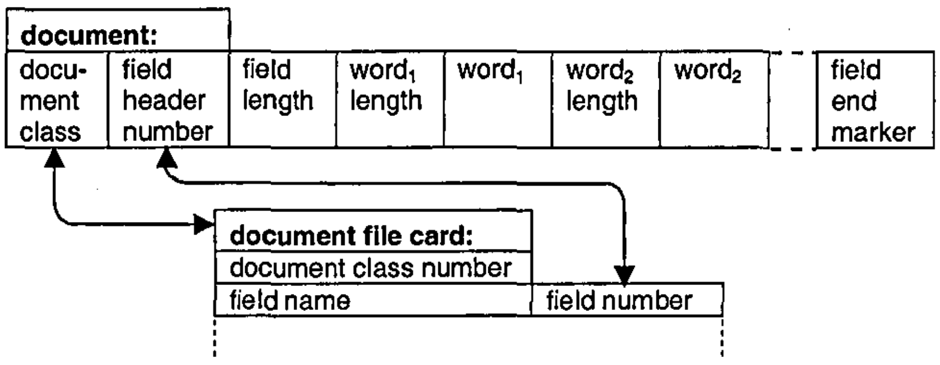


Figure 4.5. Mapping between document and document file card.

The individual fields can be nested according to the format held in the document file card. The use of binary numbers for field identifiers allows for hierarchical typing of text within a document. This structure supports queries that will vary from the generic to the specific. A search accelerator can mask the scope identifiers just as easily as masking data: it can ignore irrelevant or over-specific aspects of the categorisation. By a suitable choice of binary identifiers, specific searches can be converted into generic searches. The data is independent of any strong typing etc. as it carries its own identifiers with it.

4.4.6 Recent developments in text structuring

There has been extensive work done in text structuring in recent years and there are several standards. ODA has been mentioned earlier and there is also Structured Generalized Markup Language (SGML) [GOL90]. However, these are extensions beyond the simplified NDF format used in Textmaster and are capable of handling much more complex document structures such as text positioning, graphics, hyperlinks etc. They could easily be incorporated into our architecture if another primitive type, call it *html*, were deemed desirable.

Self-describing data formats have been in use for a long time for exchanging data between applications. Their use for exchange between heterogeneous systems is more recent. The Object Exchange Model (OEM)—part of the Object Database Management Group (ODMG)—was developed for that purpose [CAT94]. It has now become the *de facto* standard for semi-structured data. The converging standards of the world wide web (WWW) centre around the W3C—the WWW Consortium. The original mark-up language HTML (used to describe data presentation) is being merged with XML (used to describe data structure) into a new standard XHTML. There are various standards surrounding XML and the tools/parsers associated with it. A detailed description of these is beyond the scope of this thesis. However, the basic idea behind the mark-up of XML documents could equally be applied to our architecture where our documents could be structured along the same lines.

The whole area of semi-structured data is one of on-going research and is still very much in its infancy [ABI00]. The concepts behind applying database schema to semi-structured data, that we call partially structured data, is an area of continuing research. The intention is to show that meaning can be extracted from partially structured data and used to create schema extensions without changing the actual data upon which those extensions are based.

4.4.7 Pre-processing of documents

Each document in the database belongs to a document class so that a uniform format for its insertion and pre-processing will be available. It is of course possible that a document is not required to have any format or structure at all. However, this would still constitute a document class—albeit one with the minimum of structure. It would consist merely of a list of delimited words and their lengths prefixing them.

When a document is added to the database it must be done within the specifications set out for its document class. Fields such as 'heading' etc, need filling out on screen to enable the software to add appropriate identifiers. After the document has been pre-processed, the delimited words included in it need adding to the string tables—stop words excepted—together with incrementing the count of occurrences. The string triples are then extended to include the new word tokens and their corresponding relation surrogate and entity identifier.

4.5 Description and results

In order that this string architecture and enhanced searching opportunities can be assessed, we coded some search algorithms (described in section 4.5.2) and included them in two different programs using text files of people's names. We first give an overview of the basic structure, then describe the algorithms provided and the initial results.

4.5.1 Basic structure

The ability to include missing characters was built into the search algorithms as shown in the table below. Search patterns of four or more characters are required. However, a pattern like "%__%" would find all two-letter names and greater. For brevity there were some restrictions on the use of the "%" character:

the following alternatives were used as the eight search cases allowable in the programs:

case	pattern
N	"..............."
L	".............%"
M	".......%......."
ML	"......%.....%"
F	"%............."
FL	"%...........%"
FM	"%.....%......"
MM	"....%....%...."

legend	
N	none
F	first
M	middle
L	last
"."	any character
% = match zero or more characters	
"_" can be used anywhere	

Table 4.9. Search types.

The set of names (case folded) is held in two files called *forward.txt* and *reverse.txt* with the forward file structured thus:

string	token	length	occurrences
Abidi	1103527590	5	0
Abraham	377401575	7	0
Abrahams	662824084	8	0
Acres	1147902781	5	0

Table 4.10. Forward names file.

The reverse file holds the strings in reverse order i.e. "Abidi" → "idibA". The other three fields remain the same. The tokens were randomly generated.

Which string table to open is decided by examining the search pattern entered by the user. For search cases N, L, M, ML and MM the forward file is used: for search cases F, FL and FM the reverse file is used. A coarse index on each letter of the alphabet is held in memory.

4.5.2 Description of search algorithms

A brief description of the various algorithms used, which are shown in full in appendices A5 and A6, is now given. For our purposes, a display or count of the matching strings is the output: it would be equally easy to return the corresponding token for use by other functions.

Function: *fixed*

This is used for the simple case when no "%" wildcards are in the search pattern—case N in Table 4.9. The variables *start* and *stop* limit the search to the relevant part of the string table. The search is normally done from the index position of the first character in the search pattern to the start of the next character in the index. However, if the first character in the search pattern is the "_" the search is done over the entire string table. However, we note that this type of search is not likely to be requested very often. The sub function *exact* is used to match the search pattern character-by-character against each string within the index parameters *start* and *stop*.

Function: *front*

This copes with left-hand or right-hand truncation—cases L and F. The *start* and *stop* variables are set up as before and the search is done using *forward.txt* with search patterns like "mall%" or *reverse.txt* with search patterns like "%ington". After that the *search_type* variable is used to see if the pattern needs reversing before displaying.

Function: *mid*

Search patterns with one "%" wildcard embedded in them use this function—case M. After search parameters have been set up, the search looks for an exact match against each name in turn on the first part of the search pattern (up to the wildcard). If this returns true, the last part of the search pattern is matched against the last part of the same name.

Function: *midlast*

Used for cases ML and FM. Both cases are dealt with by searching for the known part of the search pattern first—i.e. "...%...%"—by dropping off the last wildcard and searching for the first part and, when successful, the second part of each string as in function *mid* above. Again, the name has to be reversed before printing in FM cases.

Function: *bothends*

Used where the search pattern has truncation at both ends—case FL—and the most expensive in terms of completion time. This algorithm discards the final wildcard and uses function *elastic* on the remaining search pattern now with format "%......." looking for a match anywhere against each name. The search through the names has to be exhaustive, as the search pattern could appear embedded in any name or in itself be a complete name (the search for "%king%" must find "King"). The function *elastic* is based on the Boyer Moore Horspool algorithm [HOR80].

Function: *midmid*

Finally, this algorithm handles search patterns like "...%...%..."—case MM. Although it was not particularly necessary to add this case of search, it was done to demonstrate how multiple cases of embedded wildcards would be handled. It is thus the most complex of the search algorithms in this section. The positions of the two wildcards are passed to the function *midmid* so that the search can proceed as follows:

```
exact match (front of search pattern, front of current name)
if (front of search pattern found)
  exact match (back of search pattern, back of current name)
  if found
    elastic match (mid search pattern, mid current name)
  fi
fi
```

The complexities of copying sub-patterns of the search pattern, and the replacement of wildcard characters, has not been discussed—see the full program in appendices A5 and A6 for complete details. The string matching algorithms used are based on the algorithms described in section 4.2.3. The same reasons apply to the choice of search strategy used as in that section.

4.5.3 Results

The number of records held in the file is 5,504,000. Each string has a unique token randomly generated for it to simulate the allocation of actual tokens that would be generated under sequential allocation. These tokens could be returned to the user but we merely return a count of the number of occurrences of each matching pattern. The times and how they scale up when using a data filter are shown in Figure 4.6 below.

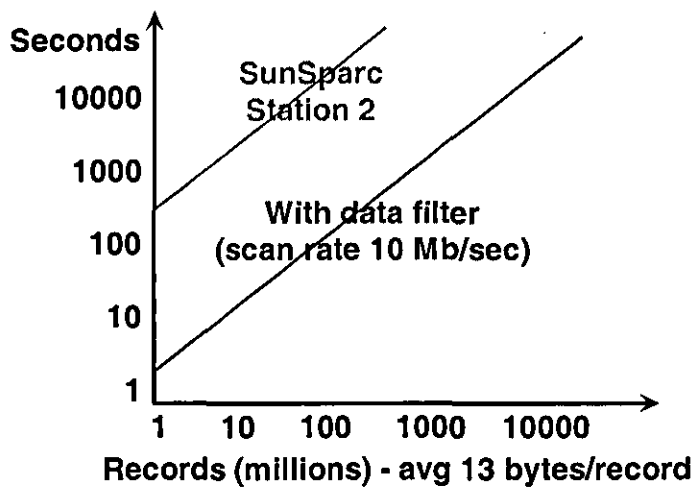


Figure 4.6. Overall string matching results.

Using some of the eight search cases described earlier, we ran searches using a Sun SPARCstation 4 on the search patterns listed in Table 4.11. These findings are also shown graphically in Figure 4.7, together with the estimated timings possible when using a data filter such as CAFS. The data filter in the example uses a scan rate of 10 Mbytes/Second. The elapsed time of the scan (where a = average record length, n = number of records and s = scan rate) is:

$$\frac{a \times n}{s}$$

Further assume that the complexity of the search pattern does not exceed the parallel processing capability of the CAFS search engine, i.e., only one search of the data is required. This boosts scanning performance and reduces search times by 90%. The timings for the first five cases include the use of a cluster index on each letter of the alphabet. The timings for the last five cases reflect a linear search.

search pattern	average time in seconds	matches found
maller	34	3,927
mall%	34	7,861
%bert	33	19,566
m%_er	40	23,434
br%e%on	53	3,910
%zz%	348	3,909
%zzz%	342	none
%qq%	323	none
%__%	340	5,504,000
_%_%	344	5,504,000

Table 4.11. Search patterns and timings.

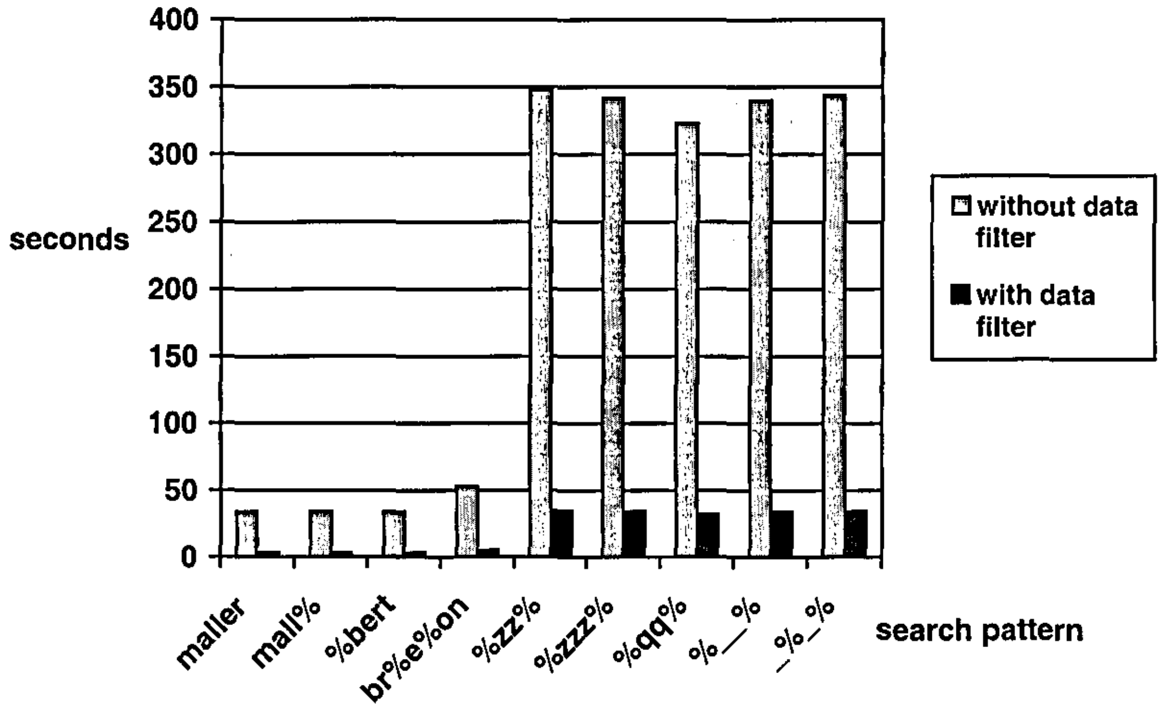


Figure 4.7. Search times with and without a data filter.

4.6 Discussion

Text handling has always been a weakness in database systems generally and relational systems in particular. The uncertainty about how to search text represents a principal difference between database management systems and information retrieval systems—such as DIALOG. DIALOG is an example of a system that makes use of inverted lists and set collation to handle user queries.

In this section we consider string handling in other functional languages and object-oriented languages. Comparisons between functional and relational languages are difficult to make and not necessarily fair. This is because relational languages (for example SQL) frequently employ complex and comprehensive indexing structures to provide rapid access, and typically use whole words as index terms.

SQL has provided string manipulating functions (*like* =, > and <>) for some time. However, the concept of large text attributes has only recently gained acceptance. In the SQL:1999 standard there is a new built-in type CLOB (Character Large OBject). The string predicate *like* can be used on instances of type CLOB. Moreover, the predicate *substring* can also be used in conjunction with *concatenate, position* and *charlength* to enable users to search lengthy texts [GRU00]. Also new to SQL:1999 is the predicate *similar* that can be used in CLOBs. This lets users construct UNIX-like regular expressions in searches, including an 'escape' character if needed e.g.

```
... SIMILAR TO 'The 7\% Solution' ESCAPE '\'
```

which would search for the film title "The 7% Solution". Although there are limitations on the use of this feature and, for historical reasons, the syntax of the regular expressions is not the same as the syntax of UNIX [MEL02].

4.6.1 Functional data languages

String manipulation is an area where functional data languages have always been weak. Functional data languages is a term used to include research into three uses of the functional approach:

- functional implementation languages
- functional query languages
- functional database programming languages.

Functional implementation languages (FILs) are usually parallel languages used to implement databases on a parallel machine. The main problem with them is to implement efficiently non-destructive updates. Notwithstanding that, the main projects—Hope$^+$ on Flagship [ROB89] and Haskell on GRIP (Graph Reduction in Parallel) [AKE93]—make no specific provision for text-intensive search operations preferring to leave these as user-definable functions only.

Functional query languages (FQLs) are used to extract information from a database but not to modify it. Most of them are included as part of a FDBPL, but not all. Some are free-standing—such as the specific language called FQL [BUN82]; some are based around a procedural language such as O_2SQL used in O_2 [BAN88]. Apart from the AGNA system (discussed in more detail in chapter 7) we cannot find examples of user functionality being passed down to the storage level. For the specific task of string matching, we are not aware of any other functional query language that supports the range of operations that we propose.

Functional database programming languages (FDBPLs) are complete languages for declaring and manipulating data. As stated above, they often incorporate a functional query language. Whereas a functional query language is only concerned with querying a database, a functional database programming language extends this by providing facilities for declaring objects (instances of classes etc.) as well as inserting and updating values. Because update is so difficult in a purely referentially transparent database, some language designers compromise the functional rules by allowing assignment to be used under certain circumstances. These are classed as *impure* functional database programming languages; examples of such are FAD [BOR90] and Galileo [ALB91]. Our system is based on a *pure* functional database programming language and thus maintains the spirit of the functional approach. However, none of these systems incorporate extensive string manipulating routines.

The omission of powerful string manipulation reflects a fundamental difference in the two paradigms: functional languages are often perceived as an academic tool devised by mathematicians with a narrow following for use as in-house teaching aids involving recursion, formal methods and compiler writing. Relational systems, on the other hand, have moved in a more commercial direction since their initial, sound mathematical specification by Codd [COD70].

4.6.2 Object-oriented database systems

The convergence and use of object-oriented (OO) concepts and database systems has not been straightforward. There has never been an agreed OO model underpinning the paradigm—as there is in the relational approach—and the many and diverse uses of OO technology has led to just as many OO database systems (OODBs). The claim—by Bancilhon [BAN88] among others—that "OODBs would become the major database technology of the 1990s" did not happen. In 1999 world-wide sales of OODB technology were less than $200 million representing a tiny percentage of the $8 billion of DBMS software sold in the same year. Moreover, while the number of relational systems has doubled since 1995, OODB systems have remained static over the same period and are in fact now in decline [WIL00].

However, although the focus of this thesis is not about the merits of OODBs, we have to consider how OODBs fare with regard to relational and functional systems. Because of the need to use methods and data encapsulation, retrieval of data can be a complex task in OODBs. In some systems—O_2 for example—the rules of encapsulation are violated to improve data retrieval. However we did not find any string matching or manipulation functions of note although of course, along with relational systems, OODBs often make extensive use of complex index structures.

4.6.3 Convergence of the functional, object-oriented and relational approaches to database systems

During this chapter we have made reference to the convergence of the functional and relational paradigms. The functional model was developed after the relational model and has not enjoyed the same commercial success. However, following the arrival of OODBs—each with its own interpretation of what the OO approach meant—the soundness of the functional model is seen by many as a better way forward and can be used to underpin the OO paradigm giving it a stricter formalism. The evolution of SQL towards object SQL has shown this to be the case with the latest standard, SQL:1999, incorporating many features from the functional model in its attempt to synthesise the object and relational paradigms.

In their paper on SQL:1999, Eisenberg and Melton [EIS99] are careful not to use the word 'object' too early in the paper due to the many connotations that the word conveys. However, many new features in the standard coincide with those from the functional paradigm, some of which have been discussed in this thesis as follows:

- Character Large OBject (CLOB) that directly relates to our *text* type. The SQL standard advocates using a surrogate in the record and holding the text separately, as indeed we do
- Boolean type is extended to include *unknown*. This is already covered by values for *unknown* and *undefined* in the TriStarp system and is being extended to include *maybe*. Other types of unknown could easily be included
- the *row* type of SQL which allows the storage of structured values in single columns—effectively violating first normal form—is easily accommodated by the functional data model where atomic values can be clustered into records for a particular entity
- recursion is included in SQL to allow for things like bill-of-materials processing. This is taken directly from the functional paradigm where is has long been used to provide expressive power
- the richness and structure of additional user-defined types are now possible in SQL. Again, these were a feature of many functional languages from the outset
- methods in SQL have a loose analogy with our integrity constraints
- both functional and dot notation is provided in SQL. For example, the expression: `WHERE emp.salary > 10,000` has the functional equivalent of `WHERE salary(emp) > 10,000`
- object identifiers (OID) are now part of the standard and bear a direct relation to the entity identifiers used in our model.

4.7 Summary

The main thrust of this chapter has been twofold:

- devolving string manipulating functionality in the current triple store architecture, and
- taking strings out of the homogeneous triple store into new data structures.

We now discuss the reasons for our choices in these areas.

4.7.1 Devolving string matching functionality

Because of the way functional languages are implemented, performance can be unacceptable for their use in practical applications—especially those that require high volumes of string manipulation with large data sets. We have shown that, by tackling string matching at a lower level in the query evaluation process, substantial performance gains are possible. Moreover, these gains come with greatly enhanced options for string matching—options that are similar to those found in traditional text retrieval systems.

As these functions are non-updating, they adhere to the basic principles of the functional paradigm—referential transparency, freedom from side effect etc. Furthermore, they can be arbitrarily nested in expressions like any other object-level function. (From an algebraic viewpoint they can be regarded as no different from the '+' or '–' operators etc.) Therefore our approach can be applied to any other comparable functional language.

A particular problem with the triple store architecture is the decomposition of strings into sets of triples, each triple holding up to six characters only. This has implications at the interface level, as the continual opening and closing of sets and crossing the interface to retrieve string triples, delays string manipulating operations. We have argued the merits of doing this before leading to the proposal to handle strings differently.

4.7.2 Handling strings differently

The rationale behind the choice of string duplication and the data structures suggested in this chapter is as follows.

First, searches can now proceed from either end of the string, as the program chooses the appropriate tables to scan depending on where any '%' wildcard characters are. The majority of search terms have either a known beginning or a known ending—this fact has been exploited.

Second, the duplication of strings complements our proposal to use RAID technology to improve data security and accessibility. The trade off is the additional space required and the overhead for updates. The additional space needed is a corollary of using RAID technology anyway and, from our design specification in section 3.4.3, is not considered an inhibiting factor. This can be achieved in the following way.

The two string tables are held in alphabetical order as described earlier. Any additions to the tables are made to the end of the file and held in a pool area for later update with the main tables. This is easily achieved using the standard UNIX *sort* command off line when, say, the entries in the pool reach a critical threshold. This means, of course, that real-time searches must also check the pool area.

The aim of using the data structures and update strategy is that they are more amenable to parallel processing and the use of a data filter. This is usually complemented by a coarse indexing structure. Data filters are best employed on sub-applications that have the following characteristics [TAG85]:

1. any input mode can apply
2. output mode is user driven rather than data driven
3. volatility is low to medium
4. data sharing is in terms of multiple queries of the same data rather than different views of the data, and
5. data structures are relatively simple.

We believe these criteria apply in our situation and justify our argument as follows.

Data filters accelerate output rather than input so it is irrelevant whether input is batch or TP (1). High volatility can detract from the benefits of using a data filter and progressively upset the match between the logical and physical sequence of a file. Once a large database has been loaded, data in the system has low volatility, so the optimum solution is to maintain a master file-set supplemented by a temporary file-set for inclusion into the main files off-line. A comparable system would be that which was developed for the UK Inland Revenue ten years ago for tracing addresses. This has 48 million records each 150 bytes in length (≈ 7.2 GB of space required.) The updates are around 5 percent per day of all records and are easily accomplished overnight (3).

Individual users should be allowed to construct their own query expressions, saving these in their own workspace where necessary in the form of macros. This is not the same as creating different logical views, where sub-sets of the same data are accessed by different indexing methods (4).

Our string data structures are of simple complexity (5). We have shown that the addition of the field identifiers to texts needed to aid scanning, is already used in commercial implementations of computer systems that use data filters. The data structures for documents are also based on well-known standards for document structuring.

The additional type *text* has been proposed so that some distinction can be made between 'short' strings used in attributes and 'long' strings used in documents. Functions can be declared, and integrity constraints designed, that take advantage of this difference—thus building extra robustness into the system.

Chapter 5 Extending interface functionality

5.1 Introduction

In chapter 3 we identified that improvements to interface functionality are needed that will permit more options for data storage and, at the same time, provide the user with more control over what can be done. In chapter 4 we introduced our object-level functions for string matching and showed how these are an improvement on the existing user-defined functions required in the current system. In this chapter we begin with an introduction to the three levels of functionality that are part of the current TriStarp system and then go into more detail on the enhancements provided [MAL98]. The three levels of functionality available are as follows.

Language level—functions declared and defined by a user written in the code of the model level language—e.g. functional. We refer to functions and functionality at this level as user-defined.

Operator level—built-in to the system, such as '+' and *substr,* for use in query expressions and user-defined functions and coded directly in 'C'. We refer to functions at this level as object-level or built-in functions.

Storage level—offers direct access to the storage sub-system, such as *string_to_token* and *fetch_another.* We refer to functions at this level as storage functions.

The first area is covered only briefly, as it does not form part of the main focus of this thesis. The second area is introduced in this chapter but is covered in more depth in chapter 7. The third area is then discussed in greater detail during the rest of this chapter, which concludes with a discussion and summary.

5.2 Extending the language level functionality

At the language level user-defined functions are declared and defined during a database session, plus the base-load functions that are loaded as part of the database environment when a new database is created. Examples of user-defined functions were given in chapter 4 and include functions like *contains* for searching for one string in another. The functions included in base-load are loaded via macro invocation. Base-load can include any user-defined function but is usually reserved for meta level functions—which include tuple and list constructors—as well as common logic and data manipulation functions. This last category includes functions for *and, or, not, head, tail, count, map* and *flatmap.* Printing and formatting functions are declared at this level as are schema functions for populating a database that can be loaded using macros at database creation.

Adding to this level of functionality is primarily a matter for the DBA responsible for creating and maintaining each database and is not discussed to any great depth in this thesis. However, there is a grey area of functionality that lies between the language level and operator level. Display functions—although not part of our area of work—could be enhanced if, for example, functions were available to the user to display all attributes of an entity. Consider the following expression used to return a list of all attributes of certain employees:

```
[{name x, ... , grade x} ‖ x ← All_emp & height x > 2]          (1)
```

With these attributes clustered on entity identifiers, an alternative expression that takes advantage of this is

```
[All_attribs x ‖ x ← All_emp & height x > 2]                     (2)
```

where *All_attribs* is a utility function which provides the same output in (2) as in (1). This is similar to the SQL *select* * operator and could be coded as a user-defined function at the language level (perhaps the best option) or included as an object-level function at the operator level (harder to do). If needed, screen formatting could also be included in the function definition. However, these are areas for further work.

5.3 Extending the operator level functionality

As well as string handling, there are other areas where functionality can be devolved from user-level to object-level that might include:

- functions used for manipulating elements of tuples and lists such as *max, min,* and *member,* plus functions to extract elements of tuples or lists such as *head* and *tail*
- logic functions such as *and, or, not,* and *if*
- miscellaneous functions such as *sum, count, average* and *id* (the identity function)
- print and formatting functions—discussed in the last section—such as *All_attribs*
- inverse functions—discussed in chapter 7
- meta functions used for querying the meta data such as *Types* and *Functions,* each of which begins with a capital letter (to distinguish them from other functions) and returns the names of the current types and functions in the database respectively.

We have in fact added to this last category by including an object-level function *Seconds* that we used to time runs of other functions. This is easily achieved by including multiple queries on one command line separated by the ';' character for example

```
Seconds; count [scp x ‖ x ← All_crm & em3 scp x "KNI%"]; Seconds;
```

would return the time in seconds before and after the result of the count expression had been evaluated so that the time it took can be deduced, thus

```
937297794
1120
937297820    ( = 26 seconds)
```

Further functions have not been coded, as the concept is one that clearly works and can be applied at whatever level the individual DBA decides is needed for each application. There is obviously more scope for inclusion of additional functionality along these lines although this is not investigated further here. The passing down of predicates, inverse functions and text functions is discussed in more detail in chapter 7, where optimisations are considered as well.

5.4 Extending the storage level functionality

The remainder of the chapter concentrates on the third level—the storage level—and what needs to be done to harmonise this with any new architecture proposals. During the remainder of this chapter the terms 'interface function', 'interface' and 'storage function' are used synonymously. In the following discussion the words 'function' and 'relation' are also synonymous: function tends to be used when describing declarations and definitions, and relation is used when discussing the function graph model and how entities and attributes are connected. The words entity and non-lexical are also synonymous.

5.4.1 Introduction

The storage level functions available to the language developers need to be enhanced to take account of the diversity of information to be stored. This will allow data storage to reflect data usage in a way that is not possible with the homogeneous triple store for all data. The original interface functions were grouped into three categories: update operators; retrieval operators and file utility operators.

The enhancements to these operators are now discussed. They aim to synthesise the original ideas from FDL [POU92] with the concepts of Hydra [KIN96b], improving upon the current interface functions provided by the BTM [DER89]. In this chapter 'storage' refers to on-line storage for the additions, etc, made during a database session. This should not be confused with the storage of data in ordered, records etc, that is done as part of the database consolidation.

5.4.2 Update operators

The current implementation provides only for insertion or deletion of a single triple, or deletion of a range of triples. However, a distinction can be drawn between operations on schema data (or meta data), and operations on instance data. The proposal for the handling of these two data types follows.

5.4.3 Schema data

Schema declarations will all fit into main memory for the duration of a session. When a new declaration is made and parsed as syntactically correct, it is copied into main memory and made persistent at the same time. It should be noted that in all the following examples the type name or function name is shown in full. The following describes how type synonyms, non-lexical types and function declarations are passed to the storage sub-system.

TYPE DECLARATIONS

The declaration of a type or type synonym by the language level user will result in the interface function *insert_type_dec(dec)* being used to record the declaration. Where dec is a pointer to the declaration; its storage is handled according to what kind of type synonym it is. For lexical-type synonyms such as: salary == real, the insertion of a single record such as

```
<"type_name", 0001, 0100, timestamp>
```

is done. Where "type_name" is the name of the new type to be used (in this case salary); 0001 is a four-bit identifier for the record shown in Table 5.1 below:

label	name	description
0001	type	type identifier
1001	prifun1	primary function identifier
1010	prifun2	multi-valued primary function identifier
1011	secfun	secondary function identifier
1100	confun	constructor function identifier
1101	icfun	integrity constraint identifier
1110	deprec	dependency record identifier

Table 5.1. Record identifiers.

and 0100 is the identifier for type real shown in Table 5.2 below

label	identifier	label	identifier	label	identifier
0001	string	0101	list	1001	function
0010	integer	0110	non-lex	1010	text
0011	boolean	0111	product	1011	expression
0100	real	1000	sum	1100	BLOB†

Table 5.2. Type identifiers.

Timestamp refers to the time the type was created. This can be held as a 32-bit number representing the seconds that have elapsed from a given start year (e.g., 1900) and is used for temporal integrity. A 32-bit number could easily hold the range of seconds that could be required, e.g., 60 (in a minute) X 60 (minutes in hour) X 24 (hours in day) X 365 (days in year) X 100 (years) = 3,153,600,000 seconds in a 100-year period. A 32-bit word size would allow for 4,294,967,296 seconds—sufficient for the above system.

For list-type synonyms like: cars == (list car), the following is inserted

```
<"cars", 0001, 0101, "car", timestamp>
```

† BLOB = Binary Large Object—discussed in chapter 7.

where the third field is the identifier for list and the type of the list—in this case *car* which has already been identified as an entity in its own right—is held in the fourth field.

Product-type synonyms such as: date == (integer ** integer ** integer) are held in one record as

```
<"date", 0001, 0111, arg_type_1, arg_type_2, arg_type3, timestamp>
```

Function-type synonyms such as: age == (integer integer integer -> integer)—which could be used to calculate a person's age when given the year, month and day of birth—are held as

```
<"age", 0001, 1001, return_type, arg_type_1, arg_type_2,
                                              arg_type_3, timestamp>
```

Extensible *sum* type declarations such as: married_status::sum are held as

```
<"married_status", 0001, 1000, timestamp>
```

where—for example—type *married_status* is the sum of three constructors (MARRIED_TO, SINGLE and OTHER) and the components of the sum type are tagged by constructors—functions without reduction rules. These return objects of type *married_status* when applied to arguments of the declared type. The constructor functions that make up the sum type would have to be stored as they are created or deleted. It would not be easy simply to append them to the declaration above. So a declaration such as

MARRIED_TO : person → married_status,

to add a person to sum type *married_status,* would result in the following addition

```
<"MARRIED_TO",1100,return_type,arg_type_1,arg_type_2,timestamp>
```

via interface function: *insert_confun_dec(dec).* Where MARRIED_TO is a constructor function with one argument which returns an object of type married_status. Constructor functions are always declared in upper case by convention.

If a facility were added to the language level so that sum types could be declared in full as one instruction, then the following kind of instruction could be held in one record as follows (where ++ means OR).

```
SINGLE:→ married_status ++ MARRIED_TO : person → married_status
                                   ++ OTHER : string → married_status
```

However, as sum types are extensible, there must be provision for addition and deletion of sum types during the current session. For this reason, it is probably better to declare the constructor functions that make up a sum type as separate instructions and store them as separate records.

DECLARATION OF NON-LEXICAL ENTITIES

A new non-lexical declaration, such as: employee::nonlex, results in the following record being entered

```
<"employee", 0001, 0110, $e, timestamp>
```

by using the interface function: *insert_nonlex_dec(dec).* $e is the reserved integer used as surrogate for non-lexical type employee in the 'is-a' triples that record instances of employees.

Chapter 5

FUNCTION DECLARATIONS

From the experience gained with Hydra, it is desirable that functions be sub-divided into primary functions and secondary functions to reflect their many differences. The main differences between these are set out in Table 5.3 below

primary functions	secondary functions
many definitions	few definitions
stores data only	used for manipulating data
only atomic values held	embedded structured allowed
only one parameter allowed	any number of parameters allowed
do not use recursion	can use recursion
able to be modified	only add new functions
dynamic	static
held in store	held in memory
must conform to function graph model	not part of function graph model
could be loaded from several files via clustering	loaded from batch file
uses best-fit pattern-matching	uses top to bottom pattern matching

Table 5.3. Contrast between primary and secondary functions.

PRIMARY FUNCTION DECLARATIONS

As far as function declarations are concerned, there is no difference between the two kinds of functions as they are both persistent. Function definitions can, however, be handled differently and are discussed in the next sub-section. Storage of primary function declarations is quite straightforward. A function such as: age : person → integer, is stored as the record

```
<"age", 1001, "person", 0110, "integer", 0010, timestamp>
```

via interface function: *insert_pfun_dec(dec),* where the second, fourth and sixth fields are type identifiers for the first, third and fifth fields respectively. This aids type checking in evaluation. Multi-valued declarations must also be catered for. A function declaration like: drives : person → list (car)—where person and car are entity types—is stored with the record

```
<"drives", 1010, "person", 0110, "car", 0110, timestamp>
```

where the second field denotes a primary function whose result is multi-valued.

SECONDARY FUNCTION DECLARATIONS

Storage of secondary function declarations can be more complex because there can be more than one argument to the function and it can return a tuple as its result. Some must also be heteromorphic in that they can be used to coerce variables to conform to parameters—in or out—that can be used with several types of argument. As an example the declaration of function *map* is as follows

map: (alpha1 → alpha2) list(alpha1) → list(alpha2)

Function *map* is the application of a function *fun* to each element of a list producing a new list in which each element has been transformed e.g.,

```
map fun [a, b, c] → [fn a, fn b, fn c].
```

The *alpha1* and *alpha2* above are type variables, e.g., `map add2 [1,2,3,4]` would use the (already declared) function *add2* to add the integer 2 to each of the numbers in the list giving `[3,4,5,6]` as the result. We believe the best way to store this is as it is actually set out, thus

```
<"map",1011,"(alpha1 -> alpha2)",1001,"list(alpha1)",0101,
                                          "list(alpha2)",0101>
```

with interface function: *insert_sfun_dec(dec)*. 1011 in the second field identifies a secondary function. 1001 in the fourth field is the *function* identifier for the third field and 0101 in fields six and eight indicates *list* type for fields five and seven.

Below are examples of how schema-level queries are handled where '?' means 'match all' and 'X' means 'don't care'.

```
Types;             /*  searches for <"?", 0001, ...>         */
Typdec "t_name";   /*  searches for <"t_name", 0001, ...>    */
Functions;         /*  searches for <"?", 10XX, ...>         */
Fundec "f_name";   /*  searches for <"f_name", ...>          */
Confuns;           /*  searches for <"?", 1100, ...>         */
Confundec "c_name" /*  searches for <"c_name", 0100, ...>    */
Ics;               /*  searches for <"?", 1000, ...>         */
```

To help with associational queries, a schema table is maintained which is inspected when using primitives such as *like.* The schema table holds the following information: Function name, Domain type, Range type, Multivalued? Number of Intensional definitions. This is shown below.

function name	domain	range	multivalued?	intensional definitions
age	person	integer	no	1
has_child	person	person	yes	0
wife_name	person	person	no	0
drives	person	car	yes	0

Table 5.4. Schema table.

The count of intensional definitions provides a potentially fast way of handling inverse graph traversal. This can be used when a function such as *All_t* is executed over type *t* returning its extent. Included in the domain there may be some instances of intensional definitions as well as the (many) extensional definitions. This is discussed further in section 5.4.6. Examples of intensional definitions include:

```
age : Fred ← age Mary; and
age : x ← 0;
```

Schema declarations are clustered around their first field and, within that, on their second field. *delete* and *modify* are handled via similar interface functions. However, the need to maintain referential transparency means amendments must be handled by a process of deletion and re-insertion. If a function requires modification, the current version must be retrieved and deleted as above, before the amended information is inserted as new triples.

5.4.4 Instance data

Before describing how the storage of on-line instance data is accomplished, it is important to emphasise the strategy regarding data storage. Under the current system persistence for both instance data and meta data is provided by an indexed storage structure based on the grid file. Meta data can be stored separately from instance data as it does not have to conform to the function graph model, nor is it needed for exhaustive searches for operations like graph traversal. However, this still leaves the bulk of the data to be stored as instance data, and this has to be stored so that the delay in responding to queries is minimised at the expense of providing dynamic, on-line update facilities.

The storage of function definitions depends on whether the function is primary or secondary. Primary functions can be further sub-divided into extensional and intensional varieties. Extensional primary functions are the more easily handled of the two because each definition has three components: <domain_name, function_name, range_name> or, to use an analogy from English, <subject, verb, object> as in: "Fred, reads, The Times". (Verb in this context is also used to include other 'joining' words like prepositions.) The update operators for: extensional primary functions, intensional primary functions, and secondary functions are now discussed.

5.4.5 Extensional primary functions

The creation of an instance of a non-lexical entity with the language level *create* command is handled as follows. Consider where a new instance of entity *employee* is needed. The command is: create employee $x, for example—where $x stands for a global variable that is allocated the next unique identifier available from the reserved area for non-lexical entities. This would result in the following 'is-a' triple being created via the interface function: *insert_non-lex_def(employee).*

```
<"is-a", "employee", $384, timestamp>
```

where 'is-a' is a reserved integer to identify 'is-a' triples, 'employee' represents the unique identifier for non-lexical type *employee* and $384 is the 32-bit integer used the represent the instance of this particular employee.

The storage of the attributes is effected by using the interface function: *insert_pfun_extdef(def)* (primary function extensional definition) where the triple might contain information such as

```
<age, $384, 0110, 30, 0010, timestamp>
```

where the third and fifth fields represent the types of the second and fourth fields respectively.

Deletion of a triple—whether for subsequent replacement by a modified triple, or simply to remove it altogether—will result in the original triple being re-inserted into the store and marked as deleted with a time stamp. The storage of multi-valued attributes, as in the expression: drives $x <= ["mini", "metro", "fiesta", "ka"], is effected using a record

```
<drives,$x,LIST,0110,0101,4,"mini","metro","fiesta","ka",timestamp>
```

where the third and sixth fields are used to ascertain the type and length of the subject component of the triple. If they did not already exist, tokens for the various cars are added to the string data structures as outlined in chapter 4.

The *include* command—for expressions like: drives $x <= include "cortina"—means the addition of a new element at the end of the record and the sixth field being incremented. The interface function for this is: *include_pfun_listdef(<subject, f_name, object(s)>).*

This results in the current record being retrieved and marked as deleted and a new record being added. The *exclude* command is handled in much the same way with the consequences for retrieval being considered in the section on retrieval operators.

5.4.6 Intensional primary functions

These are equations that involve an expression or variable (as opposed to an atomic value) on either side of their definitions. For example cases (2) and (3) below which show lexical intensional primary functions

```
age Bill <= 47;          /* extensional function */          (1)
age Fred <= age Mary;    /* intensional function */          (2)
age (x:person) <= 30;    /* intensional function */          (3)
```

There are unlikely to be many intensional primary function definitions but these forms of expressions must be catered for as they are part of the function graph model. However, there can also be intensional primary functions between entities that have the characteristics of the above three examples. For instance, assume the existence of entities *car* and *person* and a multi-valued function *drives* from person → car. Further assume the existence of person entities $p1...$p3 and car entities $c1...$c4. Then the following function permutations are also possible

```
drives $p1 <= $c3;             /* extensional function */    (4)
drives $p2 <= drives $p1;      /* same form as (2) above */  (5)
drives x <= $c4;               /* same form as (3) above */  (6)
drives $p3 <= set [$c1,$c2];  /* multi-valued (5) */         (7)
drives x <= set [$c1,$c3,$c4];/* multi-valued (6)*/          (8)
```

Bearing this in mind, the interface function to store an intensional primary function definition would take the form: *insert_pfun_intdef(def)* where *def* would constitute a record with such a format as

```
<age, "Fred", 0110, EXP, 1011, 2, "age", "Mary", timestamp>      (2)
```

where the fourth field indicates that an expression is the result of the function. The third and fifth fields are type identifiers for "Fred" and EXP, and the sixth field represents the number of space-delimited tokens that make up the function definition. All fields are fixed length. Where the function definition has an expression in the subject field the record format is

```
<age, EXP, 0110, 30, 0010, 1, "(x:person)", timestamp>           (3)
```

with a reserved value now stored as the second-last field that represents the default for this non-lexical entity class. Fields three and five are the type identifiers for non-lex and integer respectively and the 1 in the sixth field is the number of space-delimited tokens that make up the expression.

5.4.7 Secondary functions

These are not part of the function graph model so do not need to be traversed in the same way that instance data does. The current method of storing these is to use as many triples as required held as a set in memory. This means that a function definition like *map*—discussed in section 5.4.3 on schema data (secondary function declarations)—with format `map f [x|y] <= [(f x) | map` `(f) y]` is broken down into a match tree and held in memory. A detailed breakdown of how this is accomplished is beyond the scope of this thesis: good examples can be found in [POU89].

5.4.8 Retrieval operators

Recall from chapter 2 that the current system includes the following retrieval operators—where ^ indicates a pointer (call by reference) parameter:

> ***open_set(triple,^set_id)***—used to identify a set of triples which match the template *triple*
>
> ***fetch_another(set_id,^triple)***—used to return a member of the set identified by *set_id*
>
> ***close_set(set_id)***—used to close a particular set, and
>
> ***present(triple1,^triple2)***—used to determine whether there is at least one triple matching the template *triple1*. If there is, it is returned via ^*triple2*.

The enhancement of these functions is also important so that language developers understand what is available for them to use in their algorithms. There will still be several sets of records or triples open at any one time, so the interface functions *open_set* and *close_set* are still required. However, with a new storage architecture there must be a wider choice of retrieval functions to choose from when satisfying a query. The concept of a triple still applies, but now they are grouped together rather than stored separately.

As far as the language developers are concerned, they will now have a choice of what to retrieve and the granularity of record retrieval will depend on how the query expression is to be evaluated. Our hierarchy of retrieval functions is set out below where ^ before a parameter indicates call-by-reference:

get_triple(triple1,^triple2)—at the lowest level the user can request a single triple with this function. *Triple1* represents a template for one of the seven simple associative forms (discussed in chapter 2) allowable for triple look-up. If found, a triple is returned via *triple2*

get_attributes(entity,^lex_atts)—where, *entity* is a surrogate for a non-lexical entity and *^lex_atts* is a pointer to a record that contains all of the lexical attributes of entity as a list of relation-object pairs

get_Etriples(entity,^non_lex_atts)—where, *entity* is a surrogate for a non-lexical entity and *^non_lex_atts* is a pointer to all the non-lexical attributes of entity as a list of relation-object pairs

get_all_from(entity,^all_atts)—can be used to retrieve all facts about an entity as a list of relation-object pairs. This combines the previous two functions get_Etriples $\cup$ get_attributes.

Entity-to-entity triples are used in associational features, whereas the entity-to-attribute connections are considered as 'dead ends' when performing associational queries. A connection between a person aged 30 and another living in house number 30 is usually meaningless.

ASSOCIATIONAL FEATURES

Quite complex associational retrieval functions can be expressed in Hydra—see [AYR95] for a full explanation. However the basic primitives provided for the user are as follows:

from x—returns a list of all the primary functions whose domain is the same as the type of the atomic value x

to x—returns a list of inverse primary functions with the same domain as the type of the atomic value x

link n x y—returns a list of primary functions which connect the two atomic values x and y in the database, with path lengths no longer than n

trail n x y—the same as link but now returns a list of lists which set out the entities in the chain as well as the functions *like x*—retrieves all stored entities or values with the same type as the atomic value x.

The meta primitives *from* and *to* use the schema table to ascertain which entities to check—using x as the range or domain name—and retrieval of functions can then proceed. The interface functions to accomplish this are

get_all_funs_from(entity,^funs)

get_all_funs_to(entity,^funs)

where *^funs* is a list of all functions with the given entity *entity* as their range. *Link* is subsumed by *trail* and so an interface function for trail is required only. A suitable function is

get_paths(n, e1, e2, ^path)

where the first three parameters correspond to *n x y* above, and *^path* is a list of lists of relation-object pairs in the path. The use of *like* seems to have limited value as it merely returns a list of objects that share the same type signature as the parameter it receives.

5.4.9 File utility operators

The standard file utility operators are required: *ts_create*—to create a new store for a new database; *ts_open*—to open a store for use by the database and; *ts_close*—to close a database. These three functions need augmenting to reflect the new storage sub-system. So a set of functions is needed to permit the DBA to re-organise the database. These could include sorting, merging and updating indexes etc.

5.5 Discussion

Triple stores have been used in one form or another for several years now. The most recent invocation being the 'quadruple' triple store underpinning the

Sentences system [WIL00]. However, less work has been done in the area of interfaces to triple stores. The concept of a semantic-free triple store with limited interface functionality has obvious advantages but the argument for their use is not always clear cut.

The analysis of Martin [MAR84] and earlier authors, who compared semantic-free and semantic-embedding interfaces, concluded that if semantics were to be added then the triple store would, by necessity, have to be linked to the data model used. This would violate the main advantage that allows different models to use the same triple store architecture.

Our argument in this chapter (and the next) is that using a triple store for some data and a record structure for other data is perfectly possible and would not impose restricting conditions upon the model level language developers. At the model level all data is still seen as triples in the logical sense. However, at the storage level, data is physically regarded as records, triples or link files etc. The way data is retrieved is merely a matter for the storage manager to organise and is hidden from the language developer (cf. internal string triple functions described in chapter 2). It is not necessary to know if data is stored as a triple, record or in any other structure.

Furthermore, the decomposition of strings to internal triples is not one that has been followed by any other research projects that we know of. The Sentences system, mentioned above for example, uses a dual architecture comprising lookup tables for lexemes together with fixed-length tokens for triples. Moreover, the functionality of the triple store software already uses internal functions for data placement, string breakdown, etc. that do not compromise its semantic freedom. This is because they are not part of the set of visible interface functions available for the model level language developer. What we propose is an extension of these that can provide for additional functionality and optimisations at the higher level. With judicious use of semantics, a robust interface can be maintained between the language level and the storage level.

5.6 Summary

The three areas of functionality discussed can be extended in different ways. Extending the language level functionality is mainly a matter for the individual model level language developer and, for this reason, was not commented upon in any great depth. We did suggest, however, that there is room for enhancement at this level in certain areas—for instance utility functions such as *display* could be included.

We then discussed different ways (other than string manipulation) that the operator level functionality could be extended. Such ways included logic and numerical manipulators and meta functions. Finally, the main focus of this chapter concerned extending the storage level functionality to complement a new storage architecture which is discussed in the next chapter. These extensions do not compromise the semantic freedom of the interface but enhance it. Omitted from this chapter is the discussion on optimisations that are possible in passing down predicates to the storage level. This is discussed in chapter 7.

Chapter 6 An architecture to support parallel processing

6.1 Introduction

In this chapter we aim to show that, by combining mature technology with new concepts, a robust architecture can be developed that is more suited to advanced applications in our chosen domain. Achieving an architecture that permits parallel processing, is one of our overall aims in this thesis, so we discuss the NCR/Teradata DBC/1012 and the NCR/Teradata 3700 which have proven track records in management information systems, to see how their architecture would suit our purposes.

In section 6.3 we then detail our novel RAID technology that is a combination of mirroring and parity and show how data placement complements this approach. In section 6.4 we show that combining attribute records and entity triples is possible in a way that synthesises the best of the relational model with the best of the functional data model. Section 6.5 sets out our architecture giving an example of a graph traversal operation. After search strategies are discussed, we give the motivation for our choices and finish with a summary.

6.2 Using a proven parallel processing architecture design

The discussion in chapter 3 on parallel processing identified that, while there is room for algorithmic decomposition, the current architecture is not well suited to parallel implementation. In this section we consider how data can be better organised so that partitioning across independent processors can be employed. We want to spread (decluster) data across a number of disks so that a request for triples or records can be multiplexed simultaneously to the disks without concern for where the data might lie. The reasons we would want to do this are that the simplicity of the interface can be maintained and data transfer between processors and inter-process communication can be kept to a minimum.

If each processor has its own allocation of triples and records on its own disk, problems over data integrity and data sharing are minimised. This scheme is usually referred to as a loosely coupled parallel architecture and a good example of such an architecture is the NCR/Teradata DBC/1012 [PAG92]. Commonly referred to as the 'Teradata database machine' or simply 'DBC/1012', this has a proven track record in operational systems and led to the development of the NCR 3700 database machine (discussed in section 6.2.2.). A brief summary of the DBC/1012 and 3700 now follows before we show how it can be applied to our situation. Readers familiar with this may wish to proceed to a later section of the thesis.

6.2.1 The NCR/Teradata DBC/1012 database machine

The DBC/1012 is a dedicated relational database machine using a multiprocessor MIMD (Multiple Instruction Multiple Data-stream) parallel architecture. It uses standard microprocessor technology in a system that is not constrained by any particular architecture or hardware limitations. The DBC/1012 requires at least one 'host' system to connect to, and the job of the DBC/1012 is to 'off-load' from the host all work associated with relational database management and access. Figure 6.1 shows the basic components of the DBC/1012 architecture and these are now explained.

Access Module Processor (AMP)

The AMP is the database engine and manages the rows of data held in its associated disk storage units (DSU). The AMP handles all aspects of access, searching and updates of the database.

Interface Processor (IFP)

The IFP manages the flow of requests and results between the DBC/1012 and its host. The IFP accepts SQL requests from the host, chooses the appropriate sequence of actions, passes the actions to the AMPs to process, and finally returns the results to the host.

Communications Processor (COP)

The COP has a similar function to an IFP but, instead, interfaces with a LAN (local area network) to PCs, etc.

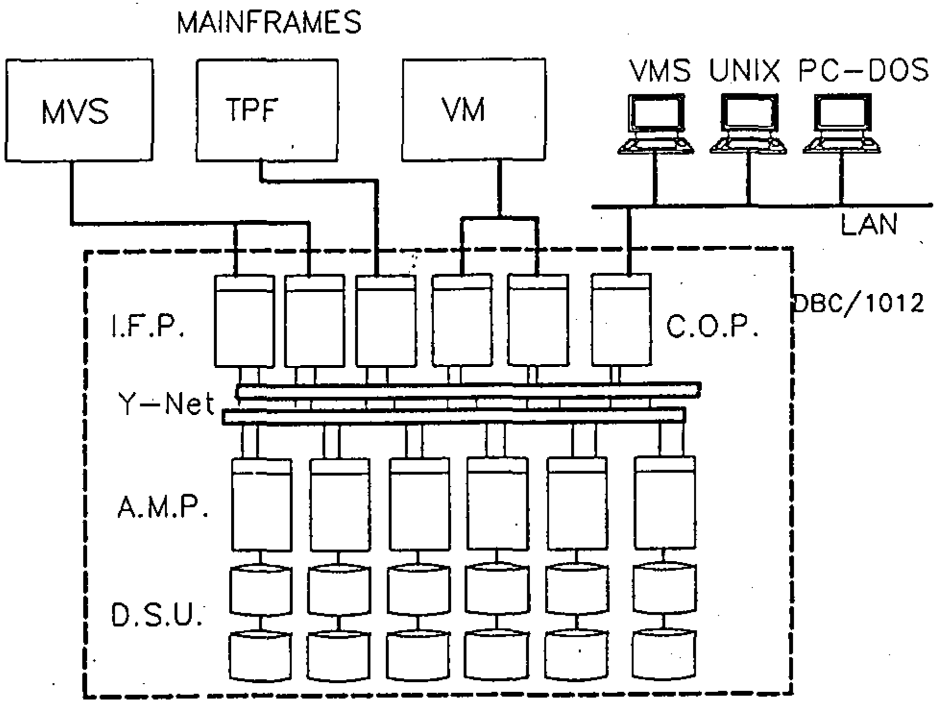


Figure 6.1. The Teradata DBC/1012 architecture.

In order that each query can be directed to each AMP, the DBC/1012 distributes tables across AMPs via a hashing algorithm. This means each AMP receives an equal number of the rows. The power of the DBC/1012 is thus directly proportional to the number of AMPs it incorporates. The hashing algorithm works on the primary index value for each row in each table and guarantees that, for example, if there are 10 AMPs, each will hold 10% of each and every table. If movement of data is required for joins, etc, this is handled by the Y-Net. For simple record retrieval the IFP passes the SQL request through the distribution algorithm that, in turn, directs the search to a single AMP.

In terms of locking, the DBC/1012 uses standard rules to permit locking at database, table and row level. As each AMP is considered as an independent computer system, multiprogramming of tasks can be easily accommodated. For data security the DBC/1012 saved records twice using two hashing algorithms to allocate rows to AMPS but recently incorporated a RAID system as standard. (We discuss our RAID architecture in section 6.3.) The DBC/1012 includes interface software for a variety of host systems—see [PAG92] for details. SQL statements may be embedded in COBOL programs where they are picked up by a pre-processor supplied by Teradata.

Software comes in two forms—DBC/1012 software and host resident software. The first is supplied by Teradata and resides in the DBC/1012: either Teradata or other vendors supply the second. (See [MAL88] for further details of this.) In addition to the standard database software, the AMPs hold software for disk and Y-Net interfaces and facilities for rollback and recovery, reorganisation and logging. Each AMP has either one or two DSUs attached to it. Principal operational use of the DBC/1012 has been for interactive management information systems covering financial services and insurance, manufacturing, federal government, telephony and retail consumer goods.

At a lower level, each AMP has a database manager (DBM) that holds two indexes for the respective DSU(s) attached to it. The *master index* contains a used cylinder descriptor list that specifies the cylinder number, table identifier and row identifier for the first row stored in that cylinder. The index is stored in table identifier order and, within that, row identifier order so that a binary search can locate the cylinder where a specific row is stored. Each cylinder has a *cylinder* *index* containing data block descriptor lists. Each of these stores table identifiers of the rows stored in a block, the row identifier of the first row in the block and the disk address of the block. They are stored in table and then row identifier so that a binary search can be used. Once the desired block is found, a sequential search through it will identify the row required (for single row entries).

6.2.2 The NCR 3700 database machine

Following the success of the DBC/1012 range of database computers, Teradata developed the NCR 3700 database machine [WIT93]. The NCR 3700 is scaled up in many ways. Processor Module Assemblies (PMA) replace AMPs and logically comprise six boards. The disk arrays use a modified RAID system and are attached to the PMAs by SCSI-2 interface. The disk array matrix is six by five disks at 1.6 GB/disk = 48 GByte of storage. Clustering of PMAs and DSU means terabyte databases are easily achievable. Protocols are extended to include Ethernet, FDDI, token ring, XNS and X25. The BYnet interconnection bus replaces the Y-net to accommodate the increases in needs. The indexing methods are a little more complex in the NCR 3700 but are essentially the same as used in the DBC/1012. There are improved joining techniques employed following the experience gained with the DBC/1012.

6.2.3 Adopting this architecture to suit our needs

Despite the many benefits of the DBC/1012 and NCR 3700, there are some issues that need to be discussed in our situation:

1. these systems only provide an interface for the relational language SQL
2. because of 1, there are limited text-handling facilities
3. for complex expressions with a low hit rate, performance would be poor

4. neither the DBC/1012 nor the NCR 3700 has to our knowledge been used in the domain of investigative systems
5. there would be difficulties in hashing records that do not have an identifiable primary key—this situation is possible with the functional data model that permits undefined or unknown information
6. entity triples do not, as such, have a 'record' identifier nor do they have a meaningful surrogate. Meaningful surrogates were evaluated in the experimental language Hydra [AYR95] but have since been deemed unsatisfactory when considering object migration
7. inverse functions are an intrinsic aspect of functional languages that may necessitate the storage of inverse triples in a functional database. Although not a feature of the relational model, this is a highly significant area of the functional data model.

For us to use the concepts from the DBC/1012 architecture, we need to consider each of these points in turn.

The interface required

Our language has its own interface for creating well-formed expressions to query the database based around the use of global variables, list comprehensions, let ... in expressions etc. There are two alternatives in overcoming the differences between relational and functional systems:

- provide a mapping from functional syntax to SQL syntax. For certain list comprehension, this can be done easily using Object SQL for example. However, functional syntax is more general (and more expressive) than SQL as it is based on functional programming. This means mappings are not necessarily so easy to achieve
- create a task manager to allocate tasks to disks directly by a series of transformations that map a user query to low-level parallel code.

Although the first option sounds easier, one of the difficulties with mapping SQL is that, because functional languages are computationally complete—they can handle recursion for example—the specification of queries is less rigid than in SQL and does not always map easily. In SQL query expressions use a small set of constructs:

```
SELECT attributes
FROM relations
WHERE conditions
AND ... etc.
```

Variations on this include using the aggregate functions such as *count, average, group by,* but these are merely 'wrapping' around the basic constructs above. In SQL allocating tasks to processors is more easily achieved. Although the second option involves a lengthier transformation process, it is possible to do this without losing any expressive power. These transformations are discussed in the next chapter.

Text handling

The nature of our domain application area means there is likely to be a great deal of text searching to be done. Significant delays in this area cannot be tolerated as a side effect of inter-process communication. Relational languages have traditionally been weak in providing powerful text handling facilities. To counter this, we include a software facility for searching text at the processor level together with enhanced functionality.

Complex expressions and low hit rates

These are other areas where a search engine can be used to assist result compilation.

Hashing where there is no unique primary key to use

The functional data model used by TriStarp allows for the creation of entities that share the same attribute. This is permitted in situations where there is little information available for a new record. Consider the situation where the following details were entered—assume entity *person* and function *name* have already been created.

```
Create person $p1;
Name $p1 <= "fred";
Create person $p2;
Name $p2 <= "fred";
```

$p1 and $p2 are global variables and are for user convenience only: in reality they refer to 32-bit sequentially allocated, unique object identifiers for instances of entity *person*. A request for information about people with name "fred" in a list comprehension might be handled as

```
[name x ‖ x ← All_person & = name x "fred"]
```

This would produce a list like [$0, $1]. This is because temporary global variables (starting at zero) are allocated for displaying entities. Other than the entity identifier used for *person* there is no primary key in this example. This is in contrast to SQL where the primary key attribute would most likely be a 'not null' field—it would always require an entry.

The standard solution to this problem is to use the object identifier for hashing the record—although with object migration this identifier might be changed at some later point. However, because of referential transparency, the complete record would require marking as deleted and a new record inserted. Therefore, allowing object migration and identifier changing should not cause difficulties. Deleting an object from one disk, and placing an amended and hashed version of it on another disk, is acceptable.

What to do with inverse functions

Inverse functions are an integral part of the functional data model marking it apart from the simpler binary relational model. They form an important part of query expressions and are frequently used for complex graph traversal operations. At the moment, they are constructed from the triple store as inverse functions but we believe that, with duplication, they can have a dual role adding redundancy to the architecture to improve security. This is discussed next.

6.3 A new RAID level

The new storage architecture is designed to exploit parallel processing techniques to boost performance, as well as incorporating RAID technology. RAID controller hardware provides data redundancy to improve reliability, either with a second, mirrored copy of the disk array—as in RAID 1 and RAID 10—or by incorporating parity information and one extra disk—as in RAID 3 and RAID 5. In addition, RAID improves performance by reducing disk bottlenecks and by increasing disk transfer rates. However, in ICL's RAID 100 mirrored system [HIL95], the redundant data is striped differently across the mirrored array. The data disk is written from the outermost track working inwards: on the mirror it is written from the innermost track working outwards. This reduces latency as data can now be read from either disk. We take this concept further by saving the entity triples in inverse function order on the mirror copy.

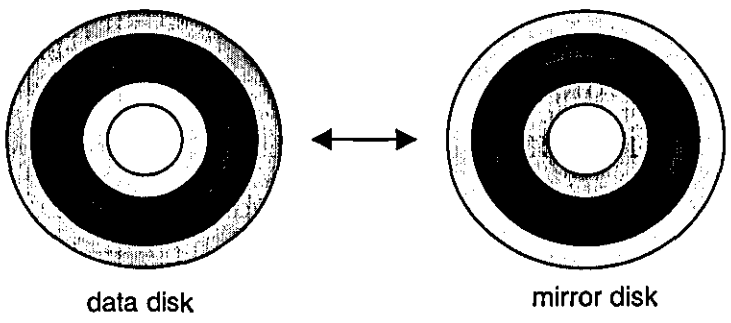


Figure 6.2. RAID 100 disk layout.

6.3.1 Our RAID approach

Our architecture will synthesise the two RAID formats to provide for even greater data security by combining the data availability of RAID 1 with the insurance of the extra disk and parity of RAID 5 as shown in Figure 6.3. RAID 5 is chosen over RAID 3 because the stripe sizes used in RAID 3 are too small. To distinguish our inverse RAID system we shall refer to it as RAID 15 (RAID 1 + RAID 5).

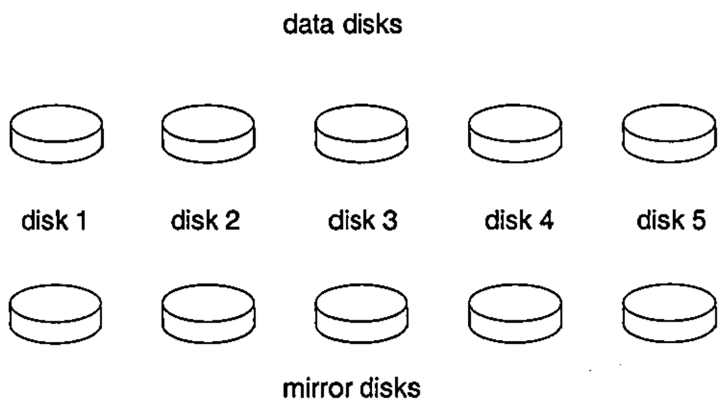


Figure 6.3. RAID 15 configuration.

The concept of combining parity and mirrored RAID systems is not new. Several authors have devised schemes that exploit the benefits of both forms of redundancy and applied these, for example, to video servers—separating the videos into 'hot' and 'cold' categories reflecting their usage [BIE97]. (We note that, although all video data uses the parity scheme, only 'hot' data is mirrored.) However, in our scheme the redundant array really is a mirror of the data array.

The performance vs. cost argument is frequently raised when considering RAID architectures. Both RAID schemes have been the subject of previous analysis, but it has often proved difficult to compare and contrast costs etc. between the two schemes. This has in fact recently been done [QUA99] and the conclusion is that neither of the two schemes can be considered as optimal in so far as performance and cost is concerned; the choice is very much application dependent. Our reason for using a 'real' mirror of the disk array is that it will complement the functional data model where the inverses of functions are frequently required for graph traversal purposes. Function inverses can either be stored explicitly or implicitly (deduced by software): we believe the former is the better option.

To illustrate this, we show an example below where the following notation holds. Function names are shown by lower-case letters, entity classes are shown by upper-case letters, and entity instances are shown by integers—the idea being that these are arbitrarily allocated from the domain of 2^{32} as actual surrogates would be. A schema table (part of) and instance graph are shown below in Table 6.1 and Figure 6.4 respectively. Then a set of triples that match the schema is used to illustrate our proposals.

relation	domain	range	MV
p	A	B	no
s	A	C	no
t	A	C	no
q	B	C	no
r	B	A	yes
u	B	D	no

(Where MV denotes whether function is multi-valued or not.)

Table 6.1. Schema table for RAID example.

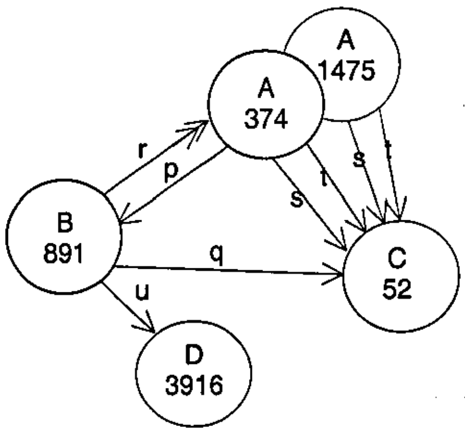


Figure 6.4. Schema for RAID example.

initial triples	sorted triples	inverse triples
<p, 374, 891>	<p, 374, 891>	<r, 374, 891>
<r, 891, 1475>	<s, 374, 52>	<r, 1475, 891>
<r, 891, 374>	<s, 1475, 52>	<p, 891, 374>
<q, 891, 52>	<t, 374, 52>	<q, 52, 891>
<s, 374, 52>	<t, 1475, 52>	<s, 52, 374>
<t, 1475, 52>	<q, 891, 52>	<s, 52, 1475>
<u, 891, 3916>	<r, 891, 374>	<t, 52, 374>
<t, 374, 52>	<r, 891, 1475>	<t, 52, 1475>
<s, 1475, 52>	<u, 891, 3916>	<u, 3916, 891>

Table 6.2. RAID triples.

The first column in Table 6.2 represents triples as they might be stored in a random or heap fashion. The second column sorts them using the following simple algorithm:

1. sort triples into entity class order — e.g. (A and B from Figure 6.4)
   (thicker horizontal line in Table 6.2)
2. within that sort into relation order — e.g. ((p, s, t), (q, r, u))
3. within that sort on subject surrogate order — e.g. (374, 1475)
4. within that sort into object order — e.g. (<r, 891, 374>, <r, 891, 1475>).

Triples are stored on the first disk array in the above order. On the RAID array they are stored after being sorted into inverse function order. This will result in the ordering found in the third column. Therefore, the above algorithm will exchange the sorts on subject and object (underlined above). This strategy enhances graph traversal operations as any expression involving entity-to-entity inverse functions, can now be targeted to the mirrored array.

Data availability is often measured in Mean Time Between Failure (MTBF) of an individual disk. The manufacturers' quoted figure for MTBF for a typical disk is 500,000 hours—about 57 years. If a mirrored system is used that has, say, an array of four data disks and four mirror disks, the MTBF can be calculated as below where 6 is the time in hours to replace a disk if the failure mechanisms are random and the probability of failure is uniform over time.

57/8 x 500,000/6 = 593,750 years.

For RAID 15 the MTBF is

(57/5 x 500,000/(6 x 4)) x (57/10 x 500,000/6) = 34 billion years

6.3.2 Improved search times

The advantage of using a RAID 15 system is as follows. Let *r* be the relation that both disks use to cluster triples (of order *n*) and let *S* and *O* be subject and object entity identifiers used in triples and search patterns. Searching for the *S* of one triple when given *r* and *O* using the data disks, means that on average *n*/2 would be inspected by linear search. Searching the <r, O, S> triples for the *S* of one triple held on the mirror disks given the same *O* and *r*, takes an average of log_2n by using binary chop search.

Retrieving a range of objects, $O_n...O_m$, using the data disks triples again requires that the whole of *r* be searched thus equal to *n*. The same range using the mirror disk would mean inspecting an average of $log_2n + |O_n... O_m|$. The increase in read efficiency is therefore the well known logarithmic improvement. However the increase in reading triples must be set against the write penalty now that the triples have to be written to two disks instead of one—although the twin disk controllers of RAID systems can handle write instructions in even time. In Table 6.3, which shows comparative search times, we have used the "70/30" rule. This estimates average throughput using the heuristic: "70% of all transfers are read transfers and 30% are write transfers". The following example compares a RAID 15 configuration with a standard RAID 5 configuration when handling 1,000 transactions (700 read: 300 write). 66 transactions per second. The searches involve locating *S* components given a suitable *r* and *O*.

	read ops	**@ 66 t.p.s.**	**write ops**	**@ 66 t.p.s.**	**total**
one array: <r,S,O>	700	10.6 seconds	300	4.5 seconds	15.1 seconds
two arrays: <r,S,O> and <r,O,S>	$\log_2 700$	.15 seconds	600 / 2	4.5 seconds	4.65 seconds

Table 6.3. Improved search times using RAID 15.

This shows that, using RAID 15, the same number of transactions can be handled in around 31% of the time that it takes using RAID 3. However, there is a cost to pay for the double redundancy of RAID 15. This is shown in the next table that compares various RAID levels using the following assumptions: disk price £1,000 each; disk speed 7,200 r.p.m.; 10 milliseconds average seek time; 1 megabyte/second transfer rate; 66 transfers/second sustained throughput with 1 millisecond overhead for each transfer. Cost of RAID controller is negligible.

RAID level	**No of disks req'd**	**MTBF in years**	**transfer rate**	**read t'put**	**write t'put**	**average t'put**	**cost average t'put**
1	8	600,000	1	528	264	449	18
3	5	71,000	4*	66	66	66	76
5	5	71,000	1	330†	83‡	256	20
100	8	600,000	1	787†	264	630	13
15	10	34 billion	1	528	264	449	22

Table 6.4. Comparison of popular RAID levels.

Additional explanation:

* — throughput of a RAID 3 system (number of transfers per second) is roughly equal to the throughput of a single disk because of the small stripe size

† — read throughput roughly equal to that for a single disk times the total number of disks

‡ — write throughput roughly equal to that for a single disk times the total number of disks divided by number of data disks

↑ — 528 times 1.49 because the outermost tracks only are read from thus reducing seek time from an average of 10 ms to around 5 ms. If rotational latency is 4.2 ms and disk transfer time is 1 ms, the saving is from 15.2 ms to 10.2 ms. The average throughput is arrived at by taking 70% of read throughput plus 30% of write throughput—as explained earlier.

RAID 15 compares reasonably well with RAID levels 1 and 5, which are the most frequently used levels, but does not come close to RAID 100. However, with the architecture used and the way triples are reversed, the concepts behind RAID 100 are inappropriate in our case. The advantages of RAID 15 are the big increase in mean time between failure, the savings shown previously in Table 6.3 and how the structure of RAID 15 complements the inverse functions that are an intrinsic part of the data model.

6.4 Combining records and triples

The Associative Data Management System (ADMS) [CRO82], introduced in chapter 3, is an example of an architecture that combines the benefits of records and triples and shows how related attributes can be collected together in sets. We want to adopt the principles from ADMS—without losing the benefits of the functional data model used by TriStarp—into a more coherent, less homogeneous architecture that will allow better use to be made of the data.

With the functional data model in mind, the following file structure is proposed. Functions between an entity and its attributes are clustered on that entity instance for storage and display purposes and referred to as **attribute records.** Functions from one entity to another entity are held as triples clustered on the subject (domain) entity class and relation name and referred to as **entity triples.** Meta data can be held as triples or records—as suggested in chapter 5—but is still referred to here as **meta triples.** These include membership or 'is-a' triples.

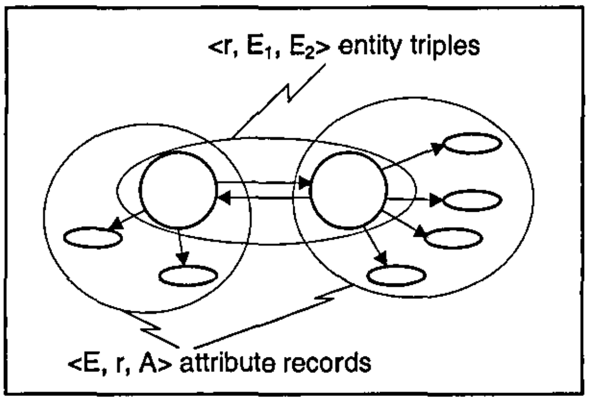


Figure 6.5. Outline of storage model.

6.4.1 Entity triples

Entity triples <r, E_1, E_2> are held in duplicate for the forward and reverse functions to which they relate. Entity triple composition was discussed in chapter 5, although omitted from that discussion was the *id_field* used for triple identification. For this field, eight bits keeps the field size as a multiple of eight bits—the same as other fields—to assist the data filter scanning software, although only four bits are required for the *id_field*:

Bit 1	dead = 1; live = 0
Bit 2	bulk = 1; non-bulk = 0
Bit 3	default variable for LHS = 1; otherwise 0
Bit 4	complex RHS = 1; otherwise 0

6.4.2 Attribute records

Attribute records <E, r, A> do not form such an important part of graph traversal operations as entity triples. They are more likely to be needed at the beginning of a search path—where a lexical value is specified—or at the end of a search path where a display is required. The records are grouped into domain sets for storage on disk in entity order with a coarse index held in memory. The structure for attribute records is as follows.

id_field: fixed length (8 bits) comprising: length of record, number of attributes in record, and one bit to indicate if record 'live' or 'dead'

entity: fixed length, multiple of 8 bits for the surrogate for this entity

relations: fixed length, multiple of 8 bits comprising a set of relation-offset pairs where relation is the surrogate for the relation name and offset is the start position of the related attribute e.g.
$<rel_1, off_1, rel_2, off_2, \ldots rel_n, off_n>$

attribute-length: fixed length, multiple of 8 bits to give the length of the whole of an attribute plus a bit to indicate if this is an intensional definition

length-word: set of length-word pairs for each individual, space-delimited word in the attribute. Length = fixed length; word = variable length. Intensional definitions are held in full text format e.g.
`{<4,Fred>,<3,Ann>}`

timestamp: same format as for entity triples.

Attributes of type *text* are held as separate link files where the attribute value held in the record is a token identifier from a sub-domain for type *text* and will 'point to' the appropriate text file. Examples of multi-valued attributes, intensional definitions and default definitions are as follows.

multi-valued: e.g. drives $d <= ["rolls", "alfa romeo"], would give the following record

```
$d,<drives,offset>,...,<A,0,{{<5,rolls>},{<4,alfa>,<5,romeo>}}>
```

intensional definitions: e.g. age $p <= age Mary, would give the following record `$p,<age,offset>,...,<A,1,{<3,age>,<4,mary>}>`

default definitions: e.g. age x <= 21, would give the following record

```
x, <age,offset>,...,<A,0,{<1,21>}>.
```

Where the *default* record is regarded as being the same as any other record.

6.5 Our architecture

Figure 6.6 shows our outline architecture. In the next sections we describe the handling of data and computation, the search and index techniques used and retrieval of data with an example.

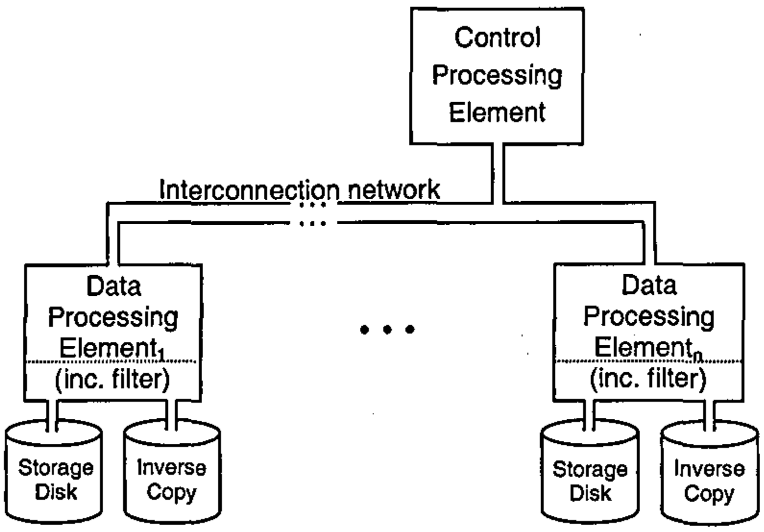

Control Processing Element
Interconnection network
Data Processing Element$_1$ (inc. filter)
Data Processing Element$_n$ (inc. filter)
Storage Disk
Inverse Copy
Storage Disk
Inverse Copy

Figure 6.6. Outline of our architecture.

6.5.1 Distribution of data

When a new entity class is created instances of it could either be clustered on the same data processing element (DPE), or be spread across the disk array in several ways—round-robin, hash, random for example. Because no semantics are conveyed in the entity identifiers, we favour a scheme that uses the last two digits of the randomly generated entity identifier to give a DPE on which to store the record or triple. This is a simple mechanism that, when used with a large enough database, will ensure a fairly uniform distribution of records and triples across the disk array.

When a search over an entity extent is needed, it can be directed to all DPEs in the disk array; when a particular record (or entity) is required the last two digits give the DPE on which it will be found. Membership triples are distributed likewise. Other meta data can be spread across the array but, as there will be relatively little of this and it is always loaded straight into memory, its placement on one disk would perhaps be better.

Lexical strings could also be randomly distributed across the disk array, where a request for a named value lookup would be dispatched to all DPEs in the array. However, if named values were mapped to DPEs via a hash value computed from the name, then the lookup operation need search only one DPE—the one to which a hash value of a name maps. Thus the lookup operation is better supported by hash distribution than by the more random nature of the record and triple placement. Hashing schemes are well documented and are not the subject of this thesis. However a hash on the last two letters of a word would generate a fairly even distribution across the array. Searches for strings with missing start or end characters need to be directed to all disks in the array.

One of the problems often put forward when discussing hashing schemes is record—or more generally object—relocation. Under our scheme, if an object changes its class this would not necessarily mean the object identifier would be changed. Therefore the DPE on which the record is situated could remain the same. The record or triple would, however, need moving from one entity class to another—i.e. *person* to *driver* class and these are stored separately on the disk.

6.5.2 Computation

Unlike some other parallel systems, we use a control processing element (CPE) on which to control the passage of a transaction through its various phases. Using a dedicated processor in this way, we can achieve locality of computation and also transaction co-ordination. All of the parallel code functions, procedures and manager macros, together with frame allocation and de-allocation (described in the next chapter), are synchronised from the CPE. This simplifies the control mechanisms needed and frees the data processing elements to handle the long-latency lookups and graph traversal operations that are the time consuming components in evaluating an expression.

An important function of the CPE is the collation of lists from the DPEs in the disk array. Using the non-strictness of the open list structures used in AGNA [HEY91], the results of each sub-list created on the DPEs can be appended together efficiently. The way this is done is by leaving the tail cell of each CONS list empty allowing the tail of *list1* to be appended to the head of *list2* using no extra storage (no intermediate lists). Finally, a *nil* value is added to the last cell of the last list. Note also that a reference must be maintained to both the first CONS cell and the last CONS cell of each list. The open list structure is similar to that for difference lists in logic programming and an example is shown below—where *v* is the value

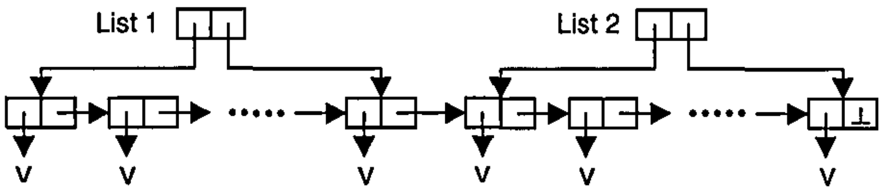


Figure 6.7. Structure of an open list.

The construction of such a list is done by using a **let ... in** block where any number of lists can be triggered in parallel with the result of the let block being the complete list. For example, if there were two DPEs, the following block that uses global variables and the *define* instruction captures the setting up of temporary lists to hold the result.

```
let   L1 == mkvarlist 1 1 @;
      L2 == mkvarlist 1 1 @;
      x₁ == define tl(L1) L2;
      x₂ == define tl(L2) nil
in L1
```

L1 and L2 are dummy identifiers set up as lists of length 1 with the element undefined. X_1 and X_2 are needed to construct extended sub-lists on the two DPEs in parallel before the result is returned. Each of these has to be passed the link values—L2 for the tail of L1 in the case of X_1 and nil for the tail of L2 in the case of X_2—as shown in Figure 6.7 above. The function *mkvarlist* is passed three parameters: the starting integer (always 1); the length of the list (in this case 1); and the type of the element (@ = undefined). This function is part of *base_load* and loaded with the schema and meta data. Its declaration and definition are as follows.

```
mkvarlist : integer integer alpha -> (list alpha);
mkvarlist x y z <= if (> x y) [] [z|mkvarlist (+ x 1) y z];
```

where `[z|mkvarlist (+ x 1) y z]` means construct a one-element list (type z) and concatenate the result of applying `mkvarlist` to the next integer in the series.

So `mkvarlist 1 4 @` would produce `[@,@,@,@]`. What actually happens in practice is captured in the following code that uses the *foldr* manipulation function to construct the lists. The variable *rest* is used to indicate the tail of each list.

```
let
   pred = filter_cond (extent theta-condition)
in
   let
      L = foldr
             (lambda (i rest) (APPLY DPE i extent pred) rest)    (1)
             nil                                                 (2)
             dpes                                                (3)
   in
      L
```

Recall that *foldr* takes a binary combining function (1), an initial value (2) and a list of values as arguments (3) and returns an accumulated value as its result. In the above, (1) is applied to each DPE as shown in Figure 6.8. The local variable *pred* is used to hold a predicate that is passed down in the *foldr* function to each DPE. The list *dpes* (3) is a list of all the DPEs in the array and *nil* (2) is the starting condition. More will be said on these optimisations in the next chapter.

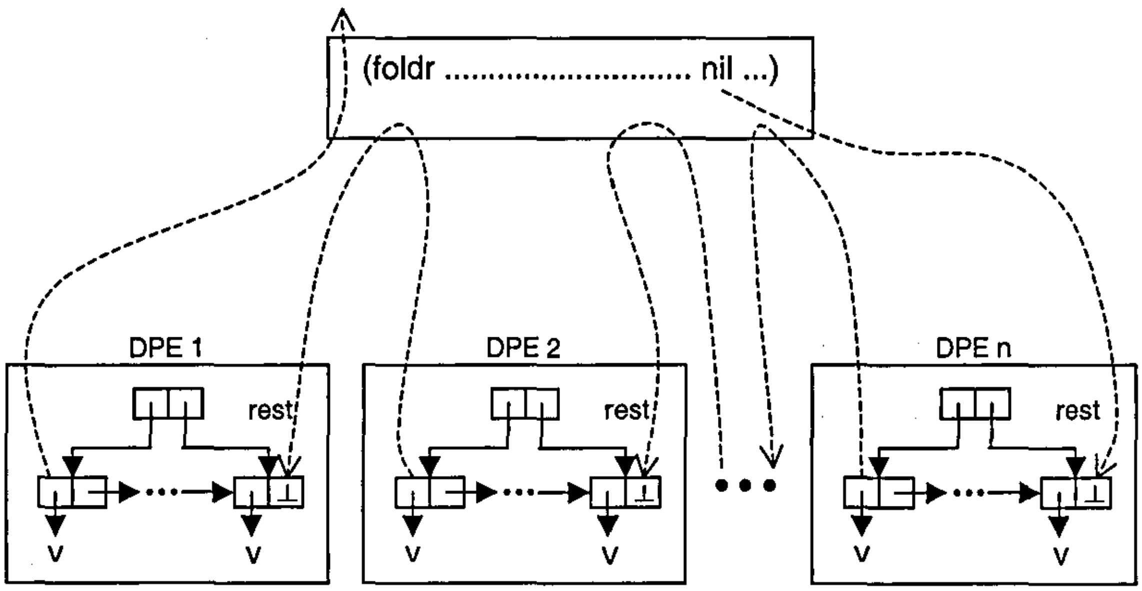


Figure 6.8. Construction of filtered lists.

By using the *foldr* function, each DPE can construct a list of results satisfying the predicate *pred* in parallel. This can be done on each DPE even though they may not have the result of the *rest* parameter. Furthermore, each DPE *i* can return the reference to the head of its particular list as soon as it is allocated. This reference is passed on in the *foldr* function as the *rest* parameter in DPE *i* – 1 where it is stored in the tail of the last CONS cell. Every DPE uses a local filter function (not shown here for the sake of brevity) to achieve this. However, the list construction is handled by expanding each local list in a similar way as was described following Figure 6.7 above. The folding and appending of the individual lists in this way means potential bottlenecks in the system, such as long latency look-ups, can be handled efficiently and safely in parallel.

6.5.3 The data processing elements (DPE)

The DPEs are used to store: the lexemes (strings) in lookup tables; the entity-to-entity triples in sets; attribute records in sets; membership triples in sets; Character Large Objects (CLOBS or documents) and Binary Large Objects (BLOBS) which are the larger attributes held in contiguous space evenly across the array. The DPEs perform the following tasks by providing mappings between:

1. lexemes (strings) and string tokens
2. string tokens and entity identifiers
3. is-a triples and entity identifiers
4. entity identifiers and attributes for display purposes.

They also handle the entity to entity graph traversal operations. The first case involves passing a string pattern—which may or may not include embedded meta characters—to the DPEs, each of which constructs its set of string tokens. After the confirmation to continue has been received from the user, the string tokens are mapped to entity identifiers (the second case). In the fourth case a set of entity identifiers is passed to the DPEs together with the functions to be displayed. For entity to entity graph traversal operations the set of triples is passed down with, either a function application *f* or inverse function application *inv_f*.

The expected speed up in execution time in loosely-coupled, MIMD, shared-nothing parallel systems is often a function of the number of data processors used to do the job. In such cases two elements are crucial in trying to achieve this: a balanced data placement policy and the search strategies employed. Data placement is discussed in chapter 8; search strategies are discussed in the rest of this chapter.

For our example, consider the mini-schema below (broken arrows show the path to be traversed) and the following (typical) query. Uppercase *A, B,* C and *D* represent the entity classes. Lowercase *a, b, c and d* represent local variables used to hold the sets of instances of the classes *A, B,* C and *D* respectively in the list comprehension. Lowercase *p* to *z* are used for function names and the '&' at the end of each line of code following Figure 6.9 means logical AND.

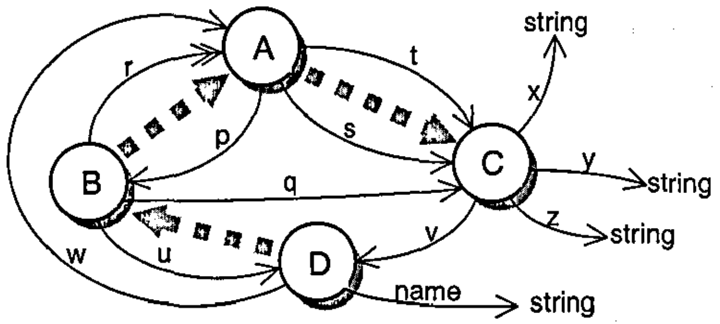


Figure 6.9. Graph traversal across DPEs

```
[(x c, y c, z c) ‖   d ← inv_name = "Fred" &
                     b ← inv_u d &
                     a ← r b &
                     c ← s a ]
```

p	374	891
p	530	4372
p	769	891
p	769	906
p	1590	109
s	374	52
s	476	79
s	530	17
s	1475	52
t	374	52
t	476	104
t	530	291
t	1475	52
q	240	98
q	758	977
q	891	52
r	87	5002
r	414	476
r	758	49
r	758	265
r	891	374
r	891	1475
u	240	3100
u	414	334
u	758	8
u	891	3916
u	1272	848
v	57	1547
v	321	42
v	1827	3916
v	8777	441
w	22	769
w	966	374
w	3100	1475
r	49	758
r	265	758
r	374	891
r	476	414
r	1475	891
r	5002	87
w	374	966
w	769	22
w	1475	3100
p	109	1590
p	891	374
p	891	769
p	906	769
p	4372	530
q	52	891
q	98	240
q	977	758
s	17	530
s	52	374
s	52	1475
s	79	476
t	52	374
t	52	1475
t	104	476
t	291	530
u	8	758
u	334	414
u	848	1272
u	3100	240
u	3916	891
v	42	321
v	441	8777
v	1547	57
v	3916	1827

Table 6.5. Triples used in single processor example.

Using the triples shown in Table 6.5, the following sequence of events would occur in a uni-processor architecture:

1. suppose the *inv_name* function returned the following set of D entities from the string tables – [8, 42, 334, 848, 1547, 3916]
2. the *inv_u* function returns the set of B entities – [414, 758, 891]
3. the r function returns the set of A entities – [49, 265, 374, 476, 1475]
4. The s function returns the set of C entities – [52, 79].

Finally, the selection of attributes for functions x, y and z is made from the attribute records for entities 52 and 79. The results are passed back to the CPE.

6.5.4 The control processing element (CPE)

This element of the architecture handles the input from users—in the form of query expressions—and the parsing, type checking, optimisation and compiling of the expression into parallel code. After that, the CPE handles any requests that need sending to the user to confirm the continuation of a query if that query is likely to take a long time to execute. The outline structure of the CPE is shown in Figure 6.10 below.

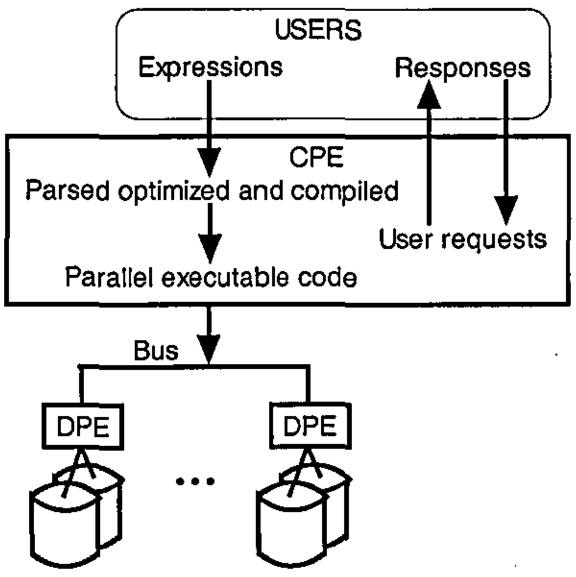


Figure 6.10. The Control Processing Element.

During the compilation of a query, the various threads of parallel instructions are concatenated to form a continuous executable code block. In the next chapter we discuss this in more detail. However, the interpretation of the executable code in which they are embedded is co-ordinated by an *interpret* manager and this is discussed next.

The selection, execution and control of instructions is managed by the CPE via the *interpret* manager which continually loops looking for an instruction to execute selecting them from the pool of active instruction threads. *Interpret* then examines the currently selected instruction and either executes it *in situ* or calls a manager function to handle this. Tasks that can be performed *in situ* include those that do not involve long latency access such as *add, sub* and *fork.* Long latency functions include *Lookup* and *StringLook.* The parameters from the program block are passed to a 'C' function corresponding to the manager name after ascertaining on which DPE the string will be found (using the hashing algorithm). Each DPE has a copy of these manager functions so searches can proceed in parallel.

If execution of a manager function involves disk access, it is not desirable for the whole process to block while waiting for this task to complete. The program can then select another thread of code to execute while the disk transfer is in progress. In such systems *load,* for example, is completed in two phases. The first phase involves starting the *load* thread and then moving on to execute other instructions; in the second phase the program is notified—by interrupt—that the I/O is complete and the result can be extracted. *Interpret* is responsible for scheduling the threads of code; those that are immediately executable from those that use a manager function. *Interpret* initiates execution of instructions that involve long latency by spawning and then enqueuing a manager request that contains the operation to be performed and its arguments, etc.

6.6 Search strategies

The point has already been made elsewhere in this thesis that our architecture does not make extensive use of indexes. Instead, our approach uses a combination of a coarse indexing structure and a search engine. A copy of the index is held on each disk in the array. For the various kinds of data that are held on the DPEs a different strategy is employed. These are now discussed.

6.6.1 Membership triples

These are spread evenly across the array so each disk will participate in the matching of class to entity identifier. More will be said about data placement in chapter 8. However, these triples are stored in their section of the disk—a section reserved for 'is-a' triples—in class order and, within that, in entity identifier order. So a triple will have the basic format as below (not all fields are shown)

```
<is-a_identifier_tag, class_identifier_tag, entity_identifier>
```

6.6.2 Other meta data

There is very little other meta data. Meta data includes: schema information; type declarations, definitions and synonyms; secondary function declarations and definitions; etc. As stated in an earlier section, these will be loaded into main memory at the start of a database session so searching for them will be relatively fast. However, persistence must be provided by writing them to the disk array, although it is unimportant exactly how this is done.

6.6.3 String tables

A simple hash function will target the search for a complete string to a particular DPE. Because of the large volume of strings to store and the randomness of the hash function a fairly even distribution can be expected. This means all DPEs will participate when a search for a string contains the meta characters that can be part of the pattern. On each DPE the strings are held in alphabetical order with a coarse index on the first letter of a string to provide the starting point for a sequential search. The DPEs can therefore be considered as 'bottomless' buckets that each takes only the strings that hash to their disk. Once a set of string tokens has been collated by the CPE and the query is to continue, the next mapping is from string tokens to entity identifiers using the string triples.

6.6.4 String triples

The string triples could also be spread across the disk array with the last two digits of a string token used to decide on which disk the string triple is located. However, once the set of string tokens has been assembled on each DPE, it would save communication time if the next mapping were performed on the same disk. Therefore, the string triples generated for each string token will reside on the disk to which the original string was mapped. Again, the CPE collates the set of entity identifiers that are then used in the graph traversal steps described earlier. Examples of this are shown in the following two figures.

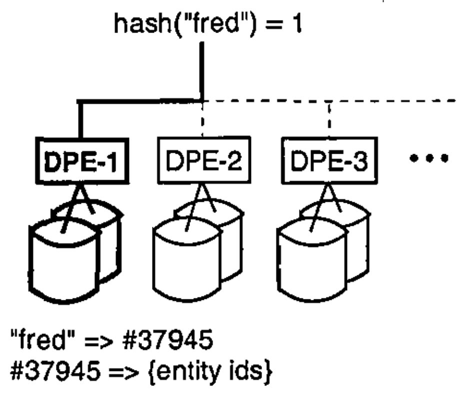


Figure 6.11. Mapping whole string.

Where there is a whole string the hash function will identify a single DPE to handle the lookup and mappings—shown in Figure 6.11 above. Where there are meta characters in the string each DPE will search its string tables. In Figure 6.12 below the string "fred%" includes the "%" character as its last element. Recall from chapter 4 that "%" means "match zero or more characters to the end of the string". So passing the search to all disks in the array means a search for strings beginning "fred...". Step (1) in the figure shows that 'hits' are achieved on DPE-1 and DPE-4 where the *#n* represents the mapping from string to string identifier. Step (2) involves assembling entity identifier sets (*a*, *b* and *c*) from the string tokens. Finally, on DPE-1 the union of the sets *a* and *b* is the result returned—step (3)—together with entity set *c* from DPE-4.

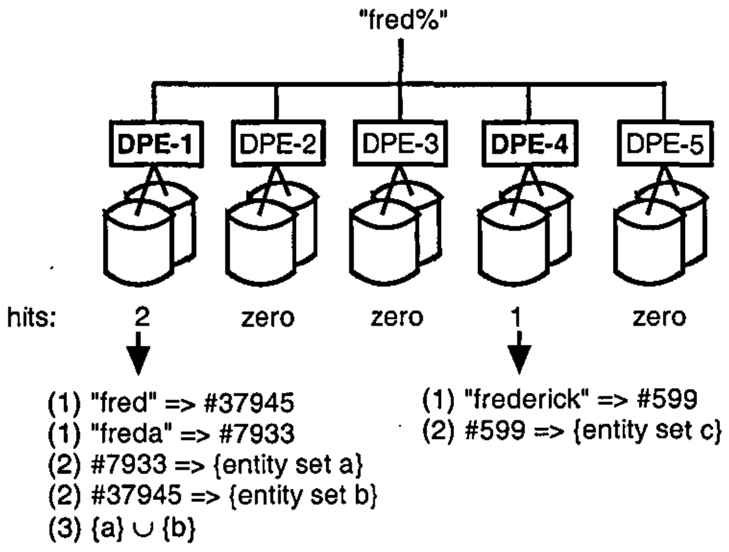


Figure 6.12. Mapping truncated string.

6.6.5 Triples and records

The last two digits of the randomly generated entity identifier will determine on which disk the triples or records will be found. Within each DPE a coarse index will identify the domain and relation name to use for a sequential search. Both entity triples and attribute records are searched in this way.

6.7 Discussion

This chapter has discussed several technologies. Multiple instruction multiple data stream machines (MIMD), redundant arrays of inexpensive disks (RAID), the associative data management system (ADMS) and the dataflow model. The first three of these are now considered. Dataflow is discussed in the next chapter.

6.7.1 The choice of a MIMD architecture

Database computers have been in existence for over 25 years but have often been confined to research applications. Of the five categories of architecture identified in [SU88], the most promising has proven to be multiprocessor database computers. A consensus on parallel and distributed database systems has emerged [DEW90] based on the shared-nothing hardware approach [STO86].

Teradata is a proven performer in the domain of large-scale applications therefore tailoring this architecture for use in the functional paradigm is a persuasive argument. Moreover, a distributed instead of a shared model of physical memory is easier to scale and allows the locality of persistent data to be exploited by use of data filters.

Other parallel functional systems have been developed over the years. Of special note must be the various parallel implementations of Haskell, a pure functional language [ARG87]. Glasgow Parallel Haskell uses the additional functions *par* and *seq* in user algorithms to instruct the compiler when to employ parallel or sequential execution. Evaluation strategies, as this approach was designated, was discussed in the earlier section 3.6.2 as was Haskell on GRIP. The Flagship project produced a parallel machine supporting declarative programming that uses the Hope$^+$ functional language. Unfortunately, the results proved disappointing [ROB89] and there were problems experienced with side effects. However, modules proved much faster to write.

6.7.2 Why RAID?

RAID is now considered as standard for the majority of new computer systems. During recent years there has continued to be improvements and enhancements in several areas. These include:

- faster drives – new 10,000 RPM drives provides improved performance over 7,200 RPM drives by 37%
- smaller RAID arrays – desktop RAID systems are now becoming commonplace
- more fibre channel – fibre channel with its 100-megabytes-a-second data transfer rate eliminates SCSI bottlenecks

Moreover, the cost of disks continues to reduce in relative terms. In 1995 [HIL95] the price-per-throughput of RAID 100 was the lowest of all RAID schemes. RAID 15 can be viewed similarly. The philosophy behind adopting RAID 15 can be summed up in the following argument. Having decided to incorporate a RAID system, why not put it to better use by making it complement the storage architecture to take advantage of the underlying data model used.

6.7.3 The storage model

The ADMS model [CRO82] and the idea of grouping like data into sets was our motivation for dividing instance data into entity triples and attribute records. More is said about the ADMS model in the next sub-section. However, we believe our architecture combines the best of the two models used—the relational data model (RDM) and the functional data model (FDM). The strengths and weaknesses of these models have been referred to in earlier chapters but the ways in which they can complement each other can be summarised as follows.

The RDM is good for grouping attributes of a named instance together, which is useful for display purposes. It does so in a 'flat' or tabular way but requires the use of complex, inverted indexing methods to achieve rapid access of attribute sets. Moreover, the joining of relations for what are in fact graph traversal operations, is complex and costly. The FDM, as originally designed, does not easily group attributes of an entity. There is duplication of entity identifiers and triples are clustered on the relation (or function) name, not on the entity itself. Thus, to display attributes of a common entity, the RDM is to be favoured.

However, the strengths of the FDM include the ability to model nested objects, to cater for a hierarchical structure of data incorporating aggregation and generalisation when needed. Entities can be migrated to higher or lower levels in the hierarchy without necessarily altering the attributes. This would be difficult to achieve using the RDM alone. The object-relational model falls somewhere between the RDM and FDM and is often viewed as the RDM with a few 'add ons'.

6.7.4 The ADMS model

Using collections of like data was part of the Associative Data Management System (ADMS) model [CRO82]. However, this model does not easily translate to the FDM so cannot benefit from the advantages the FDM has to offer. The shortcomings in the ADMS model that are alleviated by our combined model are as follows. The ADMS model does not permit more than one relationship between entities—this is of course possible and indeed desirable so real-world situations can be accurately mapped using it. To cope with this, the ADMS model has to introduce transformations of relations into dummy entity sets thus forcing the user into a particular schema design where entities exist that may not actually be required.

Integrity constraints, although included in the ADMS model, do not go as far as the integrity provision possible with our system. In ADMS this is a function of the arcs between entities and incorporates labels attached to each entity. In the FDM, the arcs represent the function (relation name) between entities and attributes. Moreover, a relationship between an entity and itself is not intuitive to model in ADMS; again, a transformation is used to introduce a dummy entity set. This is not necessary in the FDM where, for example, an entity *person* can have a *married_to* relation that refers back to *person* by simply naming a relation *married_to* that goes from *person* → *person*. In the ADMS model, for the implicit navigation algorithm to work successfully, the graph model should have the property that any two or more nodes have at least one common meeting point or upper bound. Where this is not the case, a dummy entity set is created to form a link between the two sub-entities. This is not necessary in the FDM.

For the reasons given above, we believe that our architecture—based on the FDM and incorporating ideas from ADMS—provides a more realistic way to model and then store data. Moreover, our scheme combines the benefits of the relational model (for attribute records) with those of the functional model (for entity traversal), and goes some way to achieving the four goals in the extended relational model [COD79]. In this Codd suggests that a data model should aim to have four 'personalities':

- tabular – for display and updating purposes
- set-theoretic – using relational algebra for search
- predicate calculus – for inferencing techniques, and
- graph-theoretic – to aid understanding for users and designers.

Moreover, the benefits of combining the FDM with functional programming have not been compromised. The ability to declare and store extensional and intensional function definitions is maintained. The defining of unknown and undefined values is catered for, as these can appear in any part of the database.

We have also shown how bulk data (multi-valued) attributes are handled—in both functional directions—at the entity-attribute level and for entity-entity functions. Finally, because of the semantic freedom using entity identifiers, object migration can be accommodated in our architecture. An example of this is given and discussed in chapter 8.

6.7.5 The associative model of data

A recent claim to be a radical new database model merits attention. This is the Associative Model of Data [WIL00]. This is the name given to the set of concepts, structures and techniques underpinning the *Sentences* database management system. The associative model of data builds on academic research into triple stores, semantic networks, binary-relational storage structures and entity-relationship modelling. The model is based around the concept of entities and associations (links). The associations are, essentially, the function names of the FDM. Schema are constructed incrementally—as in our model—and many of the disadvantages of the RDM are obviated using this model. Views of data are handled via permissions set in individual user profiles. These are like integrity constraints and can be defined at a low level of granularity—a single relationship (or link) is given as an example.

Chapters (rules) may be added to, or deleted from, a user's profile at any time. Individual entities and links exist in peer networks in individual chapters. When chapters are collected together into a profile, the terms and links in each chapter simply form a wider peer network and chapters become transparent. Deletions are not physically made. Instead, a 'stop' link is introduced which asserts a deleted link. Thus the association may appear to be either deleted or not according to whether the chapter containing the stop link is part of the user's profile. Re-naming is handled in a similar way.

The storage architecture underpinning *Sentences* is not clear but the indexing structure is based on the R-tree [GUT84]. Version 2 was released in October 2001 with the intention to include improved indexing techniques and a better internet interface. From the literature available it seems clear that the graphic representation of the model is not intuitive. As well as entities and links (functions in our model) there are small circles that represent 'links between links' or 'associations'. An example of a bookseller problem taken from their literature is shown below.

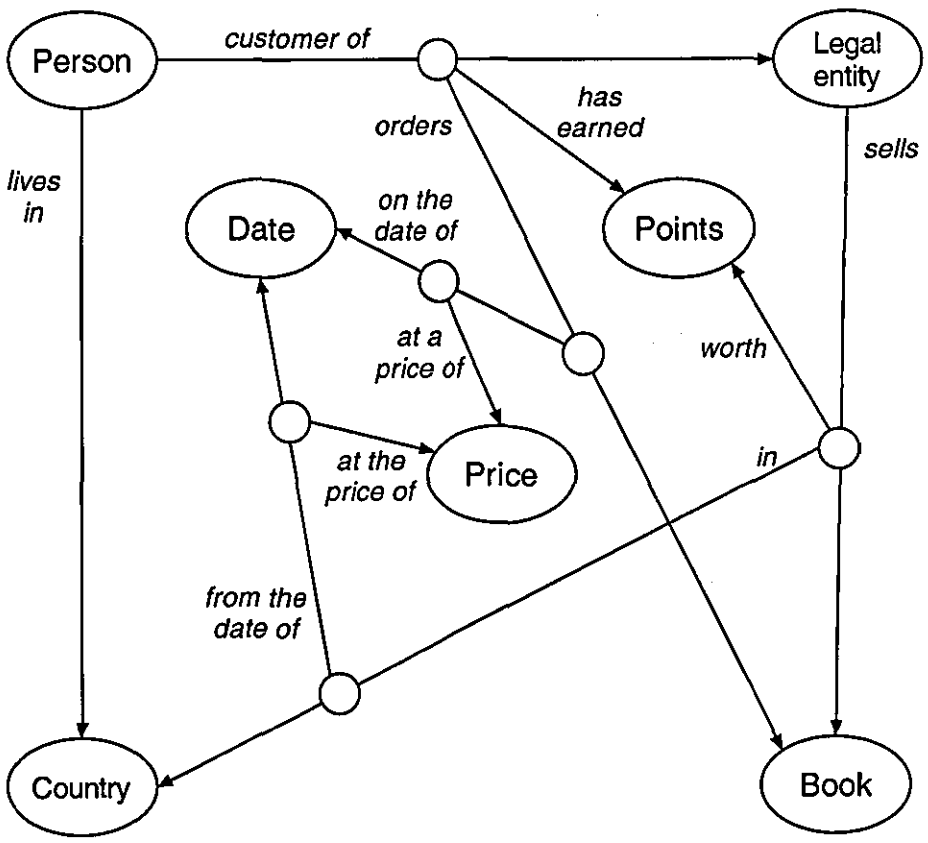


Figure 6.13. The associative model of data.

The best way of thinking about the small circles is to see them as embedded associations. The first association in the above schema would be the triple `<person, customer-of, legal-entity>`. This fact is stored by their system as—for example—identifier 231. This is then used in the association from the first circle to form the association `<231, orders, book>`. This can be viewed as a sentence with parenthesis around the sub-clauses thus: ((person is a customer of a legal entity) orders books). The database is stored in two tables: one that links identifiers with names and one that holds triples as 4-tuples—where the first field holds the identifier of the triple itself—231 in the above example. This is very similar to the FACT database system [MAG80] and obviates one of the weaknesses of binary relational models, namely, repetition of identifiers. We believe our combined model of triples and records does the same thing, is more intuitive, and retains the attributes as a record while avoiding the profusion of foreign keys found in the RDM.

It is not immediately clear how inverse functions are handled in *Sentences*. They make the point that inverses are (sometimes) important and these can be explicitly defined in the schema. However, they also use a method of implicit inverse functions where verbs (function names) such as "has" are replaced by "of" in, e.g., "has customer" becomes "customer of"—shown below.

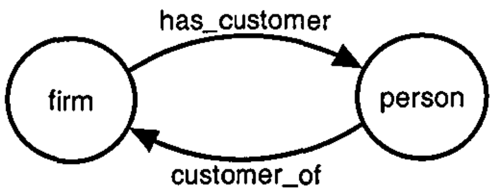


Figure 6.14. Sentences inverse functions.

Again, it is not clear how the storage of such inverses is effected. Bulk data types (multi-valued functions) are also handled obscurely. There are choices—`zero or one`, `one only`, `one or more` and `any number`—for mappings from domain to range. However, several customers use *Sentences* as a commercial product. Its share of the market will surely increase as people look for tried-and-tested commercial products—rather than academic projects—to replace the shortcomings of the RDM.

6.7.6 The Universal data model

This is another recent commercial product marketed by Universal Data Models LLC [SIL01]. It is SQL-based and provides 'off the shelf' data models (schema) solutions for a variety of business organisations. These include: health care, telecommunications, manufacturing, financial securities, insurance, service industries, travel industries and e-commerce enterprises. However, there is not a model that covers investigative systems. Moreover, as the model is aimed at relational database systems, it would seem inappropriate in our case.

6.8 Summary

In this chapter we have set out to describe and justify three elements of our architecture: using an industry proven MIMD machine configuration; harmonising the two RAID levels—mirroring and parity—to complement the inverse functions used in our model; and a physical storage architecture and data placement strategy that unifies the best of the functional data model (triples) with the best of the relational data model (records). The final element of our architecture concerns the use of dataflow graphs to distribute computation and long-latency requests between processors. This is discussed in the next chapter that deals with parallel execution and optimisation.

Chapter 7 Transformation, optimisation and translation

7.1 Introduction

In section 1.3, the background to the thesis outlined the areas of research to be investigated and the motivation behind these choices. Of the research topics discussed, parallel processing has been the least developed in the remaining chapters. This is partly because our work has added little to the existing technologies of dataflow and MIMD, and partly because no overall system has been developed on which to run accurate experiments.

However, what this chapter sets out to achieve is to show how our proposals for string handling and enhanced functionality—discussed in earlier chapters—can synthesise with a parallel dataflow architecture. In particular, in discussing parallel code, we concentrate on describing the code for our interface functions and string optimisation functions. When evaluating performance gains in section 7.6, we compare our potential improvements with the AGNA dataflow system—since this motivated our decisions to choose dataflow and MIMD—and INGRES, as comparisons can be directly made between these. The sub-sections of this chapter are now introduced.

To allow for the parallel execution of expressions, transformations and optimisations have to be made on initial user expressions. The transformation stage involves using a sub-set of the language to introduce local bindings so that sub-expressions can be executed in parallel. User expressions can often be optimised—particularly where they involve list comprehensions. We describe how standard techniques can be applied to our architecture and how to extend these to complement our text handling facilities.

After local bindings are introduced, sub-expressions are marked for reduction before the reduction process takes place. We describe the rules for the reduction process and the data structures involved. Dataflow graphs (DFGs) are then introduced and shown to fit into our architecture. These include optimisations, transformations and how DFGs are mapped to parallel code. Performance improvements are described (as outlined above) before our discussion of why we made the choices we did and, importantly, how this relates to earlier work.

7.2 Transformation to the sub-language

In order to make expressions easier to optimise and execute in parallel, the following transformations on the full language to a sub-set of the language are needed. Features that make programs easier to write but add no new expressive power to the language are removed. Also included in this stage are optimisations on list comprehensions, inverse mappings and other operations that will be performed on persistent objects in the database.

Ephemeral objects, unlike persistent objects, only exist for the duration of a transaction or database session. A user expression can be considered as a **let..in** block. The mappings are now shown where *x* indicates a new, local binding that is unique. The right arrow (→) shows the transformed expression and == means 'takes the value of'. Note that the following steps describe the process of translating user expressions to the sub-language for the purpose of potential parallel processing. They do not represent graph reduction. Graph reduction transformations are described specifically in section 7.4.1.

```
function : exp → x == (declare function exp)
```
e.g. (name : employee -> string) → x == (declare name (employee→string))

```
function exp₁ <= exp₂ → x == (define function exp₁ exp₂)
```
Used to insert information into the database e.g.,

(name $n <= "Mary") → x == (define name $n "Mary") where $n is a previously allocated global variable with an entity surrogate for a person whose name is to be defined as Mary. Also used for manipulation functions such as *map, filter* and *fold.*

```
$g == exp → $g == (exp)
```
This case can be carried straight across although (exp) may of course need further reducing.

```
type :: nonlex → x == (type nonlex)
type :: exp → x == (type exp)
```
The first case is for defining a new entity type *type*. The second case is to create a constructor function.

```
x == type → x == (type)
```
To create type synonyms such as: money == real. Again, *type* could be a constructed type.

if exp_1 exp_2 exp_3 → let x == exp_1 in (if x exp_2 exp_3)

Conditional expressions like this have to be transformed into a let block because exp_1 must always be evaluated before exp_2 or exp_3.

```
All_type → x == (All type)
```
Simply returns the current extent of type *type* to binding x that is a list.

```
inv_function op exp → x == (inverse function op exp)
```
Inverse functions can be optimised to handle other relational operators—not just equality. This is discussed in a later section. The expression `exp` has to be evaluated to give either a constant value or a non-lexical identifier before the application of this function.

```
function exp → x == (function exp)
function → x == (function)
```

Applying a function *function* to a given expression (the first case) or, where there are functions with zero arity (the second case).

```
create type $g → $g == (create type)
```

Creating an instance of a non-lexical type can be carried across using the given global variable ($g in this case).

```
delete $g → x == (delete $g)
```

Here x indicates the success or failure of the transaction.

```
op exp1 exp2 → x == (built-in op exp1 exp2)
```

Primitive function applications—like +, *div* and *matches*—can be passed across using the sub-language function *built-in*.

List comprehensions are transformed into a combination of *flatmap*, *if-then-else*, or *cons* and *nil* structures according to standard rules as in [PEY87b] for example. These functions are themselves defined in terms of other functions such as *append*, *cons*, *head* and *tail*. During the transformations, any nested functions are lifted out to the top level of the block by a process known as lambda lifting. This involves adding to its parameter list all free variables, lifting the function to a local binding with a new unique name and replacing all uses of the function by an application of it to its free variables.

7.3 Optimisations

There are well known transformations that can be applied to list comprehensions—see [TRI89] for example. The usual way this is done is by algebraic and implementation-based techniques. **Algebraic** transformations involve promoting filters in the expression so that they appear immediately after the generator with which they are used. This is especially advantageous in comprehensions that involve filtering of base extents. As an example consider the expression

```
[a ‖ (a,b)← AB & (c,d)← CD & b = c & d = 99]
```

which, after transformation, becomes

```
[a ‖ (c,d)← CD & d = 99 & (a,b)← AB & b = c]
```

The CD generator and filter is promoted before the AB generator and filter because the CD bound filter includes a constant and the equality operator.

Implementation-based techniques involve knowledge about the data such as indexes available and the size of the various extents to be searched. We do not make use of complex indexes in this way. Instead we use processor-based data filtration complemented by a coarse index structure. The size of the extent is maintained in the schema table so promotion of extents is possible. A little time spent pre-processing a comprehension has long been known to prove worthwhile in reducing execution time.

Another optimisation involves combining unary operators so that only one pass over an extent is needed. This produces an intermediary list for the selection function to use. The query

```
[sname x ‖ x ← All_student & grade x >= 7]
```

would first proceed by having a list constructed by filtering the extent (using the predicate) at the same time as the extent is traversed.

It is also possible to combine selection and filtering operations. For instance, in the above example there would be no intermediate list constructed of entity identifiers that satisfy grade >= 7. Instead what happens is that, once an entity identifier is selected it is immediately passed to the selection function to display the *sname* while the filtration over the rest of the extent is handled in parallel. This is possible because of non-strictness and the use of open lists. Each time an entity identifier is selected the result list is extended by one more CONS cell to accommodate the new identifier to have its *sname* displayed. The concept of open lists was discussed in the previous chapter.

In AGNA [HEY91], when an expression is evaluated, pre-processing of filters can identify if there are any that map over the same extent. This gives the case of multiple access paths and, if these exist, an algorithm selects the optimum route to the data based on simple heuristics and the extensive use of indexes. In our architecture, such expressions can be passed to the storage sub-system for assembling a list of entity identifiers that satisfy all filters over the same extent. Again, in the AGNA system it is possible to pass down to the storage sub-system: the extent to be used, the filter condition and the select conditions if they are over the same extent. However, this would not always be possible due to the depth of functionality that can be (and often is) applied to list comprehensions. More complex expressions often appear in the 'select' part of a comprehension—to the left of the || bar—also the qualifiers themselves can take more complex expressions. However, the concept of 'passing down' an extent generator and any filters over it is more easily controlled and is discussed next.

7.3.1 Passing down filters and generators

A scan of the expression list reveals any generators where there are filters that share the same variable. These can be 'wrapped' into a predicate condition that we call *restrict*

```
x == (restrict extent num-of-preds {function-name operator EXP})
```

where the sub-language function *restrict* is followed by the number of predicates then by an extent name—e.g. *student*—and then a number of relation-operator-expression triplets (where the expression evaluates to a constant) that are to be used in the filtering operation when list *x* is constructed. For the sub-expression

```
x ← All_student & sname x matches "Sm_th%" & grade x > 7
```

the first transformation would give

```
x == (restrict student 2 {sname matches "Sm_th%", grade > 7})
```

then the procedure would be:

1. search integer triples matching pattern <grade, 7, ?> (or greater than 7) to generate a set of entity identifiers. Call this set I-set
2. search string tables for string identifiers matching pattern "Sm_th%". Call this set S-ids
3. search string triples for matching pattern <sname, S-ids, ?> to generate a set of entity identifiers. Call this S-set
4. x == I-set ∩ S-set.

The generated set x would then be used in any further expressions such as selection of records to display. However, there is another way of passing down generators for low-level assembly of the list required that involves inverse functions.

7.3.2 Inverse functions

Currently, inverse functions can only be applied using implied equality. The generator `x ← inv_fname "Fred"` implies "the list x will hold all entity identifiers where the condition fname = "Fred" holds. This, of course, works perfectly well for other types such as `x ← inv_grade 7`. However, if we want to use other θ conditions, we are forced into using a generator and filter combination. To print the sname (surname) of all students who have a grade > 6, the expression would have to be:

```
[sname x ‖ x ← All_student & grade x > 6]
```

Therefore with equality conditions both methods can be used. The following two comprehensions were compared using the same data set taken from the crime database. (See appendices A1 and A2 for the crime database schema and triple breakdown.)

```
count [cat x ‖ x ← All_itm & cat x = "VIDEO"]          (1)
count [cat x ‖ x ← inv_cat "VIDEO"]                    (2)
```

Where (2) out-performed (1) by around two orders of magnitude. This is because (2) is evaluated at the storage sub-system level. (1) involves the generation of a list and then filtering it with the two operations controlled from the language level. The inverses of functions are not held explicitly, they are derived in the following way through software. For any first-order single-argument function f:t→s, where t is a non-lexical type and s is a non-list type, the derived function inv_f:s→(list t) is also available. The equation defining inv_f can be considered to exist as follows

```
inv_f s <= [y ‖ y ← All_t & (f y) = s]
```

A similar function is used where s is a list type. What happens is that a set of triples is opened with the pattern <f, ?, s> and the entity identifiers (y) are retrieved lazily. Because inverse functions perform so much faster than the combination of generators and filters, it would be advantageous to make more use of them. In the current system a search through triples for the third component s—where s could have any θ condition as well as = attached to it—would usually mean searching the entire extent because of the ordering of the triples.

To improve the choices available for inverse function applications, care has to be taken in matching the operator to the type of the attribute from which the inverse is taken. For instance, if the operators ">", "<" and "<>" (not equals) were allowed for strings the implications might be too severe. Whereas string operators like *contains* or *begins* could more realistically be used to select strings that contain or begin with a certain pattern. Additionally, for integer attributes the θ operators above could be acceptable but for real types they might not be. The format of the *inverse* function is

x == (inverse function-name operator EXP)

where EXP is a sub-expression that evaluates to a constant.

We handle inverse functions in two ways. If the function maps from an entity to a lexical attribute (E → A), the lexical triples provide the inverse function mapping. For integers involving range queries, a scan of the integer set will give the entity identifiers to be returned. For strings, the string tables identify the string tokens that are in turn used in the string triples. If the inverse mapping is from an entity to another entity ($E_1 \rightarrow E_2$), the mirror array provides the mapping.

7.3.3 Selection functions

Another area where a sub-language function can be used to good effect is for selecting fields of an entity that need to be displayed. Although comprehensions can have complex expressions, often involving graph traversal to the left of the ‖ bar, there are circumstances where a simplified instruction can be used for selecting attributes. If a comprehension has the form

$[\{\texttt{function}_1\ \texttt{x},\ \texttt{function}_2\ \texttt{x}\} \| \texttt{x} \leftarrow \texttt{All\_entity}\ \&\ \ldots]$

where all the selection functions are over the same variable, a *select* sub-function can be used to locate the attributes for display. Like other low-level functions that pass down expressions to the storage sub-system, a combination of the type extent plus the set of attribute functions is 'wrapped' into the sub-function *select* as follows

$$x_1 == (\text{select entity } x \{\text{function-name}_1, \text{function-name}_2, ..., \text{function-name}_n\})$$

The bound variable x is the relevant entity identifier (which could be a list). The local variable x_1 will become the list of attributes to display using the standard *print* function. The *select* function will be expanded upon in a later section that discusses the mapping of sub-language functions to code for parallel implementation.

7.3.4 High hit rates for lexical values

A fundamental part of our architecture is the ability to provide better text searching facilities. A new optimisation makes it possible for users to decide whether or not to continue with a search that is likely to have a high hit rate and therefore take a long time to execute. This is possible by checking for constant patterns to be evaluated from the string tables to identify any that will give rise to a large result set of entity identifiers.

If the user is to be given a choice of whether to continue with a search or abort it based on the number of hits found, a function must be used to obtain that information. When the total hits exceed some θ condition, the function must pass the number of hits to the user and await their instruction to proceed. The point at which this request is made has to be between the searching of the string tables and searching for string triples. This is because the string tables have a 'number of occurrences' field and it is this information that has to be passed back to the user. The pseudo code for this is shown below where dpes is the number of data processing elements available.

```
BEGIN
occs = $\sum_{i=1}^{dpes}$ occurrences(string_look_up(string_pat))$_i$
IF (occs > θ)  {
    ok = ask_user(occs)
    IF (NOT ok)
      abort transaction
}
get string triples
END
```

The two kinds of expression that contain patterns are function applications and inverse function applications. After the creation of local bindings, these two functions will be of the form:

x == (function EXP) and

x == (inverse function-name operator EXP)

where EXP evaluates to a constant and *operator* was discussed in section 7.3.2. In these cases the local variable x can therefore take the 'abort' command as well as lists of results in the normal way.

7.3.5 Text searching functions

A class of functions not discussed yet are those that operate solely on texts. Recall that attributes of type *text* are held contiguously across the disk array in a format decided upon at database set-up. The functions that operate on them can also be passed directly to the storage sub-system in the way described above, so a list of entity identifiers that satisfy the conditions can be constructed. A sub-language function, *text-fun,* initiates a search of the text attributes and has the form

x == (text-function {function-name operator EXP}) and is handled the same way as built-in function *restrict*. For example, one of the built-in functions introduced in chapter 4 was *order_str*. This takes two strings—*pat* and *text*—to see if *pat* is embedded in *text* and can have the meta character "<" in between sub-patterns of pat. This function is appropriate for use on large texts and, from the list comprehension, the mapping to the sub-language would give the binding for example

x == (text-function report order_str "flick<knife")

which would return all entity identifiers where the attribute called *report* of type *text* had the word "flick" somewhere before the word "knife". Note that these text functions are in addition to the string matching functions (previously described) that can be equally applied to text searching.

7.4 The abstract reduction machine

The preceding discussion shows how user expressions can be transformed into the sub-language and the various optimisations that can be applied. The identification of sub-expressions to be reduced is considered next. In this section we give the rules for these with an example to show how reduction can proceed and how this affects the data structures. The reduction process begins with the following starting components:

- the expression to be evaluated: EXP
- the current extent of the database: DB_{curr} and
- an update table to hold changes: T.

The new database extent can be defined as:

DB_{new} == evaluate (EXP, DB_{curr}, T)

The current extent of the database consists of a top-level environment that maps names to values in persistent store or memory, and a heap that maps identifiers to values. The heap maps ephemeral identifiers to values (those that last for the duration of a session) and temporary identifiers used for the duration of an expression evaluation only. These identifiers are allocated by the system as and when needed and are not usually visible or meaningful to users—although they can be declared as such. So, to follow previous notation, we show these as $n values. Both data structures have space for new values that will be used to update the extent if the transaction is successful. Also included in these structures are references to integrity constraints.

The update data structure records changes to type extents (insertions and deletions) and attribute values (insertion and deletion) and is used to keep track of alterations that will affect persistent objects such as integrity and meta triples.

7.4.1 The reduction process

The expression to be evaluated starts off as an initial block of expressions after type checking and bindings have been created as just described. After which the following sequence of transformations occurs. At each stage of the re-write process the next expression to be evaluated is marked for reduction with symbol $\mathcal{R}$. So the initial state of an expression is: $\mathcal{R}(let\ ((x_1,s_1), \ldots\ (x_n,s_n))\ in\ s)$. Where each x and s pair represents a uniquely bound sub-expression from the original expression. Reduction of an expression proceeds from this state until no further re-write rules can be applied. In the final state of a successful transaction, all expressions in the top-level block are reduced to values. The reductions are as follows

Declaration reductions

$\mathcal{R}$(type nonlex) → type nonlex

$\mathcal{R}$(x == type) → x == type

$\mathcal{R}$(create \$g) → create \$g

$\mathcal{R}$(delete \$g) → delete \$g

No further reduction is possible when declaring, creating or deleting non-lexical types or a type synonym. These rules do, however, entail alterations to the schema information and update tables.

$\mathcal{R}$(type exp) → type $\mathcal{R}$(exp)

Constructed types may include sub-expressions that require evaluation.

$\mathcal{R}$(declare function exp) → declare function $\mathcal{R}$(exp)

Where *exp* can refer to simple type signatures such as: *sname → string* or more complex signatures like: *(alpha1 alpha2 → list (alpha1)) → list (alpha2)* that need looking up in the top-level environment.

Definition reductions

$\mathcal{R}$(define function exp_1 exp_2) → define function $\mathcal{R}$(exp_1) $\mathcal{R}$(exp_2)

Both expressions can be marked for potential parallel execution. Exp_1 is the input parameter(s) and exp_2 the output parameter(s) as described in section 7.2 earlier in this chapter.

$\mathcal{R}$(\$g == exp) → \$g == $\mathcal{R}$(exp)

The marker is propagated to the expressions whose ultimate value will be passed to \$g.

<u>Application reductions</u>

$\mathcal{R}$(All type) → All type

$\mathcal{R}$(fun) → fun

In each case no further reductions are possible.

$\mathcal{R}$(function exp) → function $\mathcal{R}$(exp)

$\mathcal{R}$(inverse function op exp) → inverse function op $\mathcal{R}$(exp)

For a function application (or inverse) that consists of an expression, the marker is moved to that expression.

$\mathcal{R}$(built-in op exp_1 exp_2) → built-in op $\mathcal{R}$(exp_1) $\mathcal{R}$(exp_2)

$\mathcal{R}$(restrict extent exp) → restrict extent $\mathcal{R}$(exp)

$\mathcal{R}$(text-function exp) → text-function $\mathcal{R}$(exp)

Primitive function applications—such as + and *div* plus *restrict* and *text-fun*—can have the marker moved to any sub-expressions they contain. Arguments may then execute in parallel. All other arithmetic, logic and relational functions have similar reduction rules.

$\mathcal{R}$c → c

A constant requires no further reduction

$\mathcal{R}$(if exp_1 exp_2 exp_3) → if $\mathcal{R}$(exp_1) exp_2 exp_3

→ $\mathcal{R}$(if true exp_2 exp_3) → $\mathcal{R}$$exp_2$

→ $\mathcal{R}$(if false exp_2 exp_3) → $\mathcal{R}$$exp_3$

For conditionals the above rules ensure the first expression is evaluated and reduced to a Boolean value before the second or third expressions are reduced.

The outline pseudo code for the reduction algorithm can be considered as follows:

```
input (expression_list)
var count = 0
var next[MAX_STR]

BEGIN
    WHILE expression_list NOT empty
       next = expression_list[count]
       if (reducible(next)) mark_for_reduction
       increment count
       END-WHILE
END
```

where the procedure *reducible* uses the given rules. The expression list is the user expression after it has been reduced to the sub-language, described in section 7.2, and optimisations have taken place.

7.4.2 A reduction example

Finally in this section we give an example of how a transaction would proceed. The following three expressions—to locate a person called Fred and change the name to Freda as well as define the age as 21—are transformed as follows. (Note the propagation of the marker $\mathcal{R}$ is shown from a visual perspective and in parallel. This saves time and space but is not necessarily how it would actually occur.)

```
$n == inv_fname = "Fred";
age $n <= 21;
fname $n <= "Freda";
```

After type checking, this is transformed to a let ... in block with local bindings naming x_1, x_2 and result and the reduction marker showing there are reductions to be done.

```
ℛ(let $n == inverse fname = "Fred";
      x_1 == define age $n 21;
      x_2 == define fname $n "Freda";
      result == x_1 ∧ x_2)
in result
```

Next the ℛ marker is propagated to the three sub-expressions that have now been bound to local identifiers.

```
ℛ(let $n == ℛ(inverse fname = "Fred");
      x_1 == ℛ(define age $n 21);
      x_2 == ℛ(define fname $n "Freda");
      result == ℛ(x_1 ∧ x_2))
in result
```

The next stage propagates the ℛ to all sub-expressions of these three expressions:

```
ℛ(let $n == ℛ(inverse ℛfname = ℛ"Fred");
      x_1 == ℛ(define ℛage ℛ$n ℛ21);
      x_2 == ℛ(define ℛfname ℛ$n ℛ"Freda");
      result == ℛ(ℛx_1 ∧ ℛx_2))
in result
```

At this point there are no further reductions possible. So the next step is to map top-level names to tokens. Note we have assumed for this example that the name "Fred" is unique. The first step involves replacing constants with the tokens that represent them. This gives:

```
ℛ(let $n == ℛ(inverse ℛfname = T-Fred);                    (1)
      x_1 == ℛ(define ℛage ℛ$n T-21);
      x_2 == ℛ(define ℛfname ℛ$n T-Freda);
      result == ℛ(ℛx_1 ∧ ℛx_2))
in result
```

where T-... indicates a token mapping. Top-level function names can now be converted to tokens giving:

```
ℛ(let $n == ℛ(inverse T-fname = T-Fred);                           (2)
        x_1 == ℛ(define T-age ℛ$n T-21);
        x_2 == ℛ(define T-fname ℛ$n T-Freda);
        result == ℛ(ℛx_1 ∧ ℛx_2))
 in result
```

We now have to evaluate the first expression to obtain a token for $n before the second and third expressions can continue:

```
ℛ(let $n;                                                          (3)
        x_1 == ℛ(define T-age ℛ$n T-21);
        x_2 == ℛ(define T-fname ℛ$n T-Freda);
        result == ℛ(ℛx_1 ∧ ℛx_2))
 in result
```

This now allows us to bind $n in expressions two and three with the newly allocated token giving:

```
ℛ(let $n);                                                         (4)
        x_1 == ℛ(define T-age $n T-21);
        x_2 == ℛ(define T-fname $n T-Freda);
        result == ℛ(ℛx_1 ∧ ℛx_2))
 in result
```

Finally the two remaining expressions can be reduced to tokens and the result can be returned. Although in this example there is no visible output to the user screen, the binding of the user-defined global variable $n to the token has been successfully completed.

This transaction also changes the top level environment in the following ways. Assuming a simple environment for the moment, the changes are:

name	token
fname	T-fname
age	T-age
Fred	T-Fred
...	

➔

name	token
fname	T-fname
age	T-age
<Fred>	<T-Fred>
Freda	T-Freda

Note that lexical names are taken from the lexical tables and are only discarded when there are no other bindings for them. The integer 21 does not need an explicit mapping: integer tokens are inferred by simple bit manipulation. In this case we have assumed the name "Fred" does not occur elsewhere in the database so can be deleted (shown as <Fred> above). The heap would have a mapping for the user-defined global variable $n to link it to the entity identifier that has *fname* function mapping it to Fred (now Freda). The two intermediate variables, x_1 and x_2, would not be stored as they were not requested by the user. Their only purpose is to hold the results of sub-expressions that are evaluated along the way to the ultimate result. After that, any resources they required are freed back on to the heap. The triples to be inserted and deleted are:

triple	time	insert/delete
<T-fname, $n, T-Fred>	<32-bits>	delete
<T-fname, $n, T-Freda>	<32-bits>	insert
<T-age, $n, T-21>	<32-bits>	insert

Note that not all information held in triples is shown in the above table and that the $n only represents the token for the entity identifier that is used in persistent store. At this stage integrity constraints are checked before binding new names to identifiers and printing out any results. Only then can the transaction be closed. Committing updates, which can be done at any time by the user, causes updates to become persistent.

During the transformation process there are several places where parallel execution of expressions could take place. In (1) and (2) above, both the looking-up of top-level names and the mapping of constants to identifiers could be done in parallel. As soon as $n is defined, occurrences of it in other sub-expressions could also be defined (3). Lastly, the two defining functions in (4) could be performed in parallel. After all sub-expressions have been marked for reduction, the next step is to transform the various sub-expressions into dataflow graphs for possible parallel execution.

7.5 Translation into dataflow graphs

This section begins with an introduction to dataflow graphs. Readers familiar with the basic concepts involved may wish to proceed to section 7.5.1 where optimisations are discussed.

Instead of using a simple graph reduction process, data can be thought of as dynamic, flowing through a collection of passive transformers (the operators). Each operator performs some task on the data, as they become available on input arcs and passes the result to other operators via output arcs. This computational model is defined in terms of dataflow rather than control flow or graph reduction and programs are thus represented as dataflow graphs (DFGs). DFGs have some advantages over the other models; one of which is that DFGs are acyclic. This means that, once an operator has consumed input on its input arcs and passed on the results via its output arcs, the resources it used can be discarded. This greatly simplifies garbage collection. Once DFGs have been constructed, there is scope for parallel execution of sub-expression within the overall query. The basic dataflow machine can thus be said to be data driven, implementing eager evaluation using the call-by-value computational rule.

DFGs are particularly appealing for use in declarative languages because they do not require complicated dependency analysis. Traditional languages are usually tied to an imperative model where partitioning of instructions into fine-grain threads for parallel execution makes control far more complex. It has been shown [ARV88] that compilers for declarative languages can extract orders of magnitude more parallelism than is possible with traditional languages. It is possible to modify the DFGs to a demand-driven version that permits lazy evaluation using the call-by-need computation rule. However, this makes control mechanisms significantly more complicated.

The various constructs of a programming language can be represented graphically by DFGs where the nodes represent operators and the arcs represent data dependencies. Data driven means nodes 'fire' when their required input arcs are available. Parallel implementation is easily achievable using this model with a referentially transparent functional language.

After the initial transformations to the sub-language, an expression will have all its functions named—via lambda lifting—and defined at the top level with local bindings. The basic operators used in DFGs are: *primitive function, copy, value, fork, merge* and (the most complex) *apply*—see [FIE88] for full descriptions of these. As a simple example of a DFG, the expression (x + y) * (x – y) might have the graph

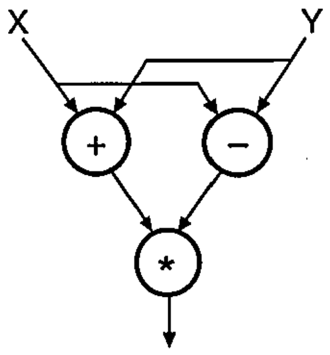


Figure 7.1. Arithmetic DFG.

The graph consists of three instructions each with an opcode, two input arcs and one output arc. Data values are carried on tokens that flow from the output arc of one instruction to the input arc of another instruction. Instructions only execute when their 'firing rule' is satisfied. The firing rule for strict operators, such as + above, states that the instruction may fire only when both inputs are present.

Execution of an instruction consumes input—sometimes producing side effects such as creating a new persistent value—and creates output. DFGs capture all the fine-grain parallelism of the source language and make explicit any data dependencies and multiple use of a variable. The + and – operations may execute serially or in parallel—it makes no difference. However, the * operator depends on the + and – operators and so cannot fire until both its input arcs have tokens placed on them.

Some instructions, such as *constant*, have no normal input values. But, without input values there is no way to give a firing rule. To resolve this, trigger tokens are added to the DFG. These carry no meaningful value but are used to initiate execution of an instruction. For such instructions the rule states that the operation should execute when its trigger input token is available—plus any inputs that are needed.

A related problem is what to do with outputs of operations that are not actually required for the result. These include triple additions and deletions that are made as a side effect of an expression but not actually consumed in any way by other instructions. While such outputs do not contribute directly to the result of the expression, it would be useful to know when they have completed and are available if required. The answer is to collect such outputs together into a 'complete' instruction which issues a completed signal token when its inputs are all fired.

Although signal tokens carry no meaningful value, once they are fired (along with the result return output) they enable all computation in a DFG to terminate successfully. In our system the DFG in Figure 7.2 shows how triggers and signals fit into the overall DFG design. The expression

```
$n == inv_fname = "Fred";
age $n <= 21;
fname $n <= "Freda";
```

would first be transformed to

```
let   $n == inverse fname = "Fred";
      x₁ == define age $n 21;
      x₂ == define fname $n "Freda")
      result == x₁ ∧ x₂
in result
```

then the DFG

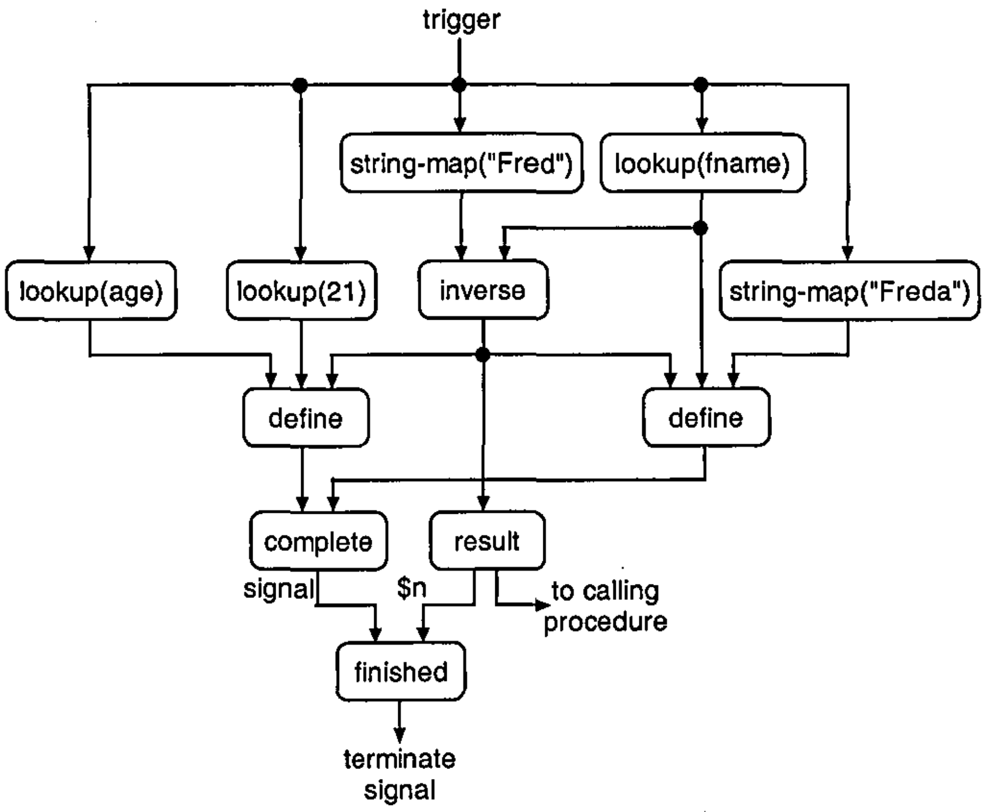


Figure 7.2. Example of a DFG procedure.

This graph shows what happens to the *signal* and *result* tokens. The *result* can be returned to the calling procedure as soon as it is available: the *signal* token is issued when all other activity in the graph is complete. The *finished* operator collects the *signal* and *result* outputs and passes a *terminate* signal to the caller. Resources can then be freed by the callee.

The above graph can be considered as a lambda (nameless) function or procedure for the purposes of controlling a sequence of transactions. The lambda instruction is connected to the graph in which it is embedded via the *trigger* and *terminate* signals. The firing rule is: when the *trigger* is activated, the procedure is created on the heap and a token carrying a reference to it is placed on the *result* output. In the graph, *define* and *inverse* are built-in operators that directly accesses the storage sub-system once their arguments have been provided. Note that in our architecture a function is applied to all of its arguments (full application) not a partial (or curried) application. Conditional expressions, such as if-then-else, are a special case and are handled using the following DFG.

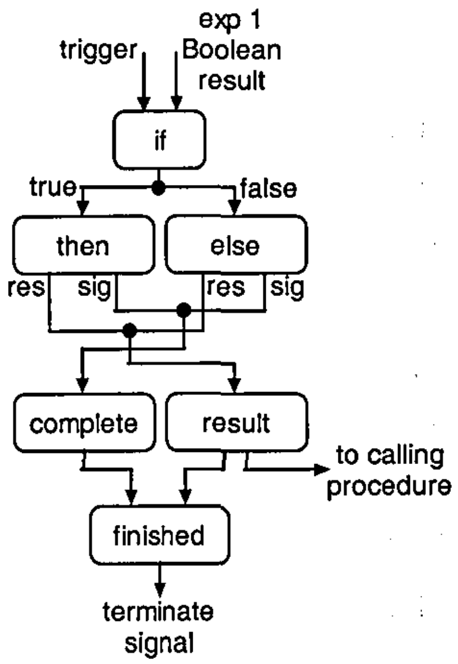


Figure 7.3. Conditional DFG.

When the result of *exp*$_1$ is known either the *then* or the *else* path is selected. When either of these has successfully completed, the *signal* and *result* outputs are activated accordingly. However, in this case similar outputs from both branches are joined using the *merge* operator before becoming the inputs to the *complete* and *result* operators. This ensures that either branch can fire the *complete* and *return* triggers thus embracing the semantics of conditional operators. Any expression or part of an expression that requires sequential execution is ordered in a sequential manner so that execution does not proceed in parallel.

7.5.1 Optimisations to DFGs

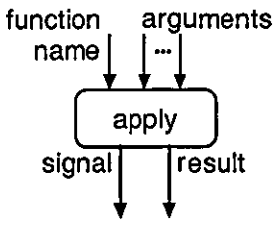


Figure 7.4. DFG for *apply*.

The most complex DFG operator is *apply*. This is needed for user functions and is shown above. Each invocation of a user function forces the creation of a new *apply* operator with its own function name and argument inputs. Application of user functions are embedded in the *apply* operator where the non-strictness of applications is embodied in the firing rule. Because of non-strictness, a CONS cell can be returned before the evaluation of the arguments. This is in keeping with dataflow schemes, so we can immediately pass back the CONS cell while the rest of the list is being assembled. After each *apply* completes, the result is returned back to the caller *apply* and, ultimately, the initial starting expression block. Even using tail-recursion, the final result has to pass back along the chain of *applies* before a result can be returned.

One of the optimisations employed in AGNA involved altering the *apply* construct to handle tail recursion better. This was achieved by propagating both the result and termination signal of each nested function call forward to the next iterative *apply* instead of back to the calling *apply*. A new signal input is added to each call to ensure that a call invoked via *tail-apply* does not complete and send a termination signal prior to its caller completing.

DFGs used in our system are enhanced by the inclusion of operators to complement our architecture; *define* and *inverse* were mentioned above. The standard DFG operator *lookup* is used to map a bound variable or constant name to a token. For names that are part of the meta data—function names, types, and global variables—this is easily achieved using the schema tables held in memory. Mapping non-string types to tokens is trivial too: simple bit manipulation turns an integer or real into a token. However, mapping strings to tokens is more complex because of the number of strings used and the possible use of meta characters embedded in them, and can give rise to large token sets. We also want to give users the choice of aborting a query if there is a high hit rate. To facilitate these features, a new DFG operator is introduced to complement the standard *lookup* operator. We call this operator *string-map* to reflect its use.

When it has been fired, *string-map* searches the string tables to accumulate a set of string tokens that match the pattern. When this has been done, a count is made of the total occurrences there are of words matching the given pattern in the database. This total is passed to the user if it exceeds a threshold asking for confirmation to continue with the query. If the query is to proceed, the set of string tokens is used to accumulate a set of entity tokens for the inverse function mapping. This new operator was used in the earlier DFG.

How do we know when to use *string-map* and when to use *lookup*? Lookup is used for non-string cases—meta data names, relations etc: the *string-map* operator is used for string constants including those with embedded meta characters. This implies there may be more than one instance of the pattern that would generate a list of string tokens. In the DFG shown in Figure 7.2, the two string constants "Fred" and "Freda" are handled by the *string-map* operator as they are strings and might, in other circumstances but not in this, generate a list of results.

Built-in functions

Recall from the introduction to DFGs that there is a *primitive function* operator. This is used for built-in functions such as +, –, *div* etc which often come at a level beneath user functions. This category could also cover our range of built-in string matching functions discussed in chapter 4. However, as outlined in the previous discussion, the patterns involved in handling strings can be such that a large set of string tokens is assembled for the mapping process. For built-in functions that do not operate on strings the operation proceeds as follows. Consider the simple predicate `age x > 21`. The DFG for this would be

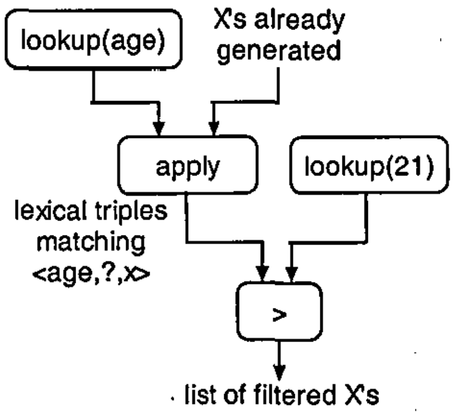


Figure 7.5. DFG for *built-in* function.

The above shows how the *apply* function generates a list of triples matching the template `<age, ?, x>` for the third component to be used in the comparison with 21. For straightforward string expressions (including those with embedded meta characters) the situation is different. We use a function *contains* that essentially means the pattern "must be contained in" the attribute. So the predicate, `fname x contains "Fred%"`, used with the following DFG, uses set intersection to obtain a list of string identifiers, then entity identifiers, before the *contains* comparison is made.

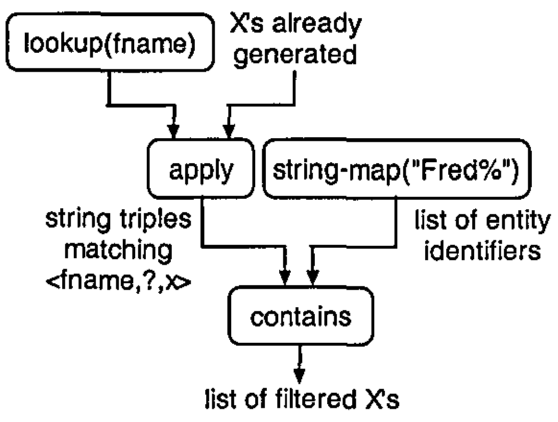


Figure 7.6. DFG for string lookups.

Here the *apply* operator is used to generate a set of Xs that may have already been reduced by an earlier function application. Therefore the *contains* operator will ensure only those entity identifiers that are relevant will get through. Furthermore, note the *string-map* operator also embodies the *apply* operator. This is discussed in the next section on translation to parallel code. However, there is often a better way to handle such functions. Where generators and filters range over the same extent, the *restrict* operator may be more applicable.

The sub-language function *restrict* is used to handle generators and filters bound to the same variable and passes them to the storage sub-system for the creation of the result list. The format of this function is

x == (restrict extent num-preds {function-name, operator, EXP})

where EXP evaluates to a constant. Once this has been done, the *restrict* function can be applied. Thus, for example, the outline DFG for the expression: `restrict student 2 fname contains "Fred%" age > 21`, is shown below.

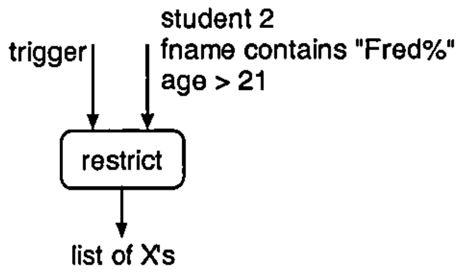


Figure 7.7. DFG for *restrict.*

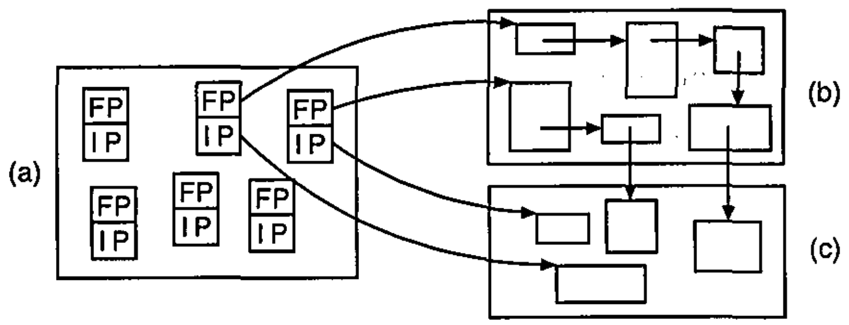
Note that the low-level functions *restrict, inverse* and *text-map* do not use *apply* to evaluate. Nor do they use *lookup* and *string-map* in the same way as user functions do. Instead their list of arguments is passed directly to the storage sub-system to compile a result list once any embedded expressions in their graph have been evaluated. The looking up of strings is handled differently because of the need to pass conditions directly to the disk array to arrive at a filtered list of entity identifiers. The "abort" condition can also be returned from here if the user does not wish to continue with the query.

Finally, the *select* function is used to select attributes of an entity for display purposes. The DFG for this function is similar to those for *restrict* etc, in that the inputs are the trigger and the condition list and the output is a list of items to display using a standard *print* instruction. The next stage involves transforming DFGs into machine level instructions and organisation of parallel threads of code.

7.5.2 Translation to parallel code

In this section we discuss our parallel abstract machine proposals. In the following three pages we give a brief outline of this topic setting out the standard code. We pay particular attention to the novel areas that are specific to our architecture. The fine-grained threads that underpin the parallelism are allocated and managed by a **parallel machine.** The parallel machine consists of a pool of active thread descriptors and separate memories for frames and the heap. Each thread consists of an **instruction pointer** (IP) and a **frame pointer** (FP). IP points to the current instruction residing in the code of a heap-based procedure call; FP points to a frame.

Frames are allocated and de-allocated as part of procedure call and return, and provide local storage for arguments and computation. Organised into a tree, there is a frame for each procedure call. Multiple frames can be active simultaneously and each frame can have many active threads. The parallel machine proceeds by extracting an active thread, executing its current instruction and adding to the thread pool any necessary descriptors. The execution order for threads is not specified and multiple threads may execute concurrently.



(a) Thread descriptors, (b) Frame memory, and (c) Heap memory.

Figure 7.8. Organisation of the parallel machine.

Memory is divided into persistent store—storage held in the disk array—and heap memory. The heap is used for persistent objects—triples, records, meta data—that are added to the database extent after successful termination of a transaction, and transient objects—variables, lists, etc—used for the duration of a database session or duration of a transaction.

The semantics of parallel instructions are in terms of state transitions on frame memory, heap memory (including persistent store) and the pool of thread descriptors denoted (FP, IP). The instructions required for the parallel machine include the standard operators for reduction machines which are found, for example, in [HEY91]. (In all cases r, ri and rj refer to offsets.)

Control flow

```
▪  jmp L - add descriptor (FP, L)
▪  jmc r L - if frame[FP+r] = 0 add descriptor (FP, L)
             else add descriptor (FP, IP+1)
▪  fork L - add descriptors (FP, L) and (FP, IP+1)
▪  die - add no descriptor
▪  join r bn - if bit n of frame[FP+r] set
               add descriptor (FP, IP+1)
               else add none
               toggle bit n of frame[FP+r]
```

Join is used to combine and synchronise parallel threads creating a successor descriptor only if the *join* bit is set to one. For example, in the code below threads T1 and T2 are combined and synchronised by the join instruction at L1:

```
T1:   r1 ← x
      jmp L1
T2:   r2 ← y
L1:   join r3 b0 - wait for x and y
      add r4 r1 r2
      ...
```

T1 and T2 place x and y into frame slots r1 and r2 and transfer control to L1. T1 transfers control via jmp, whereas T2 just "falls through" where the join bit is initially set to zero. When join is first executed (by T1) the bit is set to one and no new successor descriptor is added—the thread is terminated. When T2 executes join, the bit is set back to zero and the thread continues with the addition.

Arithmetic, logic and relational operators

```
▪  binop r1 r2 r3 - frame[FP+r1] = frame[FP+r2] binop frame[FP+r3]
▪  unop r1 r2 - frame[FP+r1] = unop frame[FP+r2]
▪  loadc r1 c - frame[FP+r1] = c        :where c is a constant
```

In each case a new descriptor, (FP, IP+1), is added.

Heap access

- `load r1 r2 - frame[FP+r1] = heap[frame[FP+r2]]`
- `store r1 r2 - heap[frame[FP+r1]] = frame[FP+r2]`

In each case a new descriptor, (FP, IP+1), is added.

Inter-frame transfers – call and return

- `fcall r1 r2 -` caller frame initiates callee frame

```
let FP' == frame[FP+r1]
let IP' == frame[FP+r2]
add descriptors (FP, IP+1) and (FP', IP')
```

- `fret r1 r2 r3 r4 -` callee frame transfers result to caller

```
let FP' == frame[FP+r1]
let IP' == frame[FP+r2]
let r == frame[FP+r3]
let v == frame[FP+r4]
frame[FP'+r] = v
add descriptors (FP, IP+1) and (FP', IP')
```

In addition to these standard instructions, there are several longer, macro-style instructions for frequent database operations. These include the following.

- `AllocObject ri rj -` allocate & initialise new object on heap

```
let type == frame[FP+ri]
let size == frame[FP+ri+1]
let addr = allocate block of heap memory for size
frame[FP+rj] = addr
add descriptor (FP, IP+1)
```

- `AllocFrame ri rj -` allocate and initialise new frame

```
let CFP == frame[FP+ri]         :CFP = caller's FP
let RIP == frame[FP+ri+1]       :RIP = result IP
let SIP == frame[FP+ri+2]       :SIP = signal IP
let Res == frame[FP+ri+3]       :Res = where result stored
let slots == frame[FP+ri+4]     :number of slots required
let FP' == new frame & set slots to zero
FP' = (CFP,RIP,SIP,Res)
frame[FP+rj] = FP'
add descriptor (FP, IP+1)
```

- `MakePersist ri rj -` move object from heap to persistent store

```
let obj == frame[FP+ri]
frame[FP+rj] = obj
```

```
frame[FP+rj+1] = false
if volatile(obj)
  if alreadymoved(obj)
    frame[FP+rj] = lookupPerAdd(obj)
  else
    let A == address in persistent store
    copy(obj,A)
    frame[FP+rj] = A
    frame[FP+rj+1] = true
add descriptor (FP, IP+1)
```

- `DeAllocate ri` - frees heap storage held at address

```
let obj == frame[FP+ri]
addtofreelist(^obj, length(obj))
add descriptor (FP, IP+1)
```

Then there are macros pertinent to our architecture. These include

- `InsertNonlexDec` – insert non-lexical declaration into update table T
- `DeleteNonlexDec` – mark the above as deleted
- `InsertPriFunDec` – insert primary function declaration into update table T

where table T is part of the heap. These match the interface functions introduced in chapter 5. As an example, consider the third macro *InsertPriFunDec* used to add the triple

```
<"age", 0010, "person", 0110, "integer", 0010, timestamp>
```

to the table T. In this triple, the second field identifies the type of the triple (primary function definition), the fourth and sixth fields are the types for the third and fifth fields. The macro to add this is

```
InsertPriFunDec ri rj   let rel == frame[FP+ri]      :relation name
                        let tty == frame[FP+ri+1]    :triple type
                        let sub == frame[FP+ri+2]    :subject name
                        let sty == frame[FP+ri+3]    :subject type
                        let obj == frame[FP+ri+4]    :object name
                        let oty == frame[FP+ri+5]    :object type
                        let sec == frame[FP+ri+6]    :timestamp
                        if (ok)
                          frame[FP+rj] = ()
                          add (rel,tty,sub,sty,obj,oty,sec) to T
                          add (FP,IP+1) to thread pool
                        else return error
```

The other procedures relevant to our architecture are those that deal with passing down generators and filters, inverse functions and selection. These require new loader instructions to set up the frame memory.

```
loadr r1 exp - to load restrict expressions
loadi r1 exp - to load inverse expressions
loads r1 exp - to load selection expressions
```

The *loadr* instruction sets up the required frame offsets to store the extent name, the number of predicates and the predicates themselves. The *loadi* function sets up the required frame offsets for the extent—inferred from the function name—the function name, the operator and the value. The *loads* function sets up the frame memory with the extent, list of entity identifiers and list of functions to use for retrieving attribute values. As an example *loadr* is handled as follows

```
Loadr ri exp
      frame[FP+ri] = exp[1]                :extent
      frame[FP+ri+1] = exp[2]              :num preds
      for (j = 1; exp[2] x 3; j += 3)
        frame[FP+ri+1+j] = exp[2+j]        :function
        frame[FP+ri+1+j+1] = exp[2+j+1]    :op
        frame[FP+ri+1+j+2] = exp[2+j+2]    :value
```

This uses the number of predicates held in the expression to control how many offset places are needed to hold the three elements of each predicate. The *Lookup* procedure is set out below.

```
Lookup ri rj
      let name == frame[FP+ri]
      let token == get token for name from meta triples
      if (found)
        frame[rj] = token
        add descriptor (FP, IP+1)
      else return error
```

This uses the meta triples to establish and return a token bound to the top-level name. The procedure for *string-map* is more complex and uses the manipulation function *foldr*: this is discussed below.

```
String-map ri rj
      let name == frame[FP+ri]
      let fun == frame[FP+ri+1]
      let (occ,stoks) == foldr (k rest) APPLY dpe k string-match name
                               (0, nil)
                               dpes
      in
            if (occ > θ) AND (NOT ok=ask_usr(occ))
                  frame[rj] = abort
            else let eids == foldr (m rest) APPLY dpe m open-s fun stoks
                             nil
                             dpes
                 in
                       frame[rj] = eids
                       add descriptor (FP, IP+1)
```

This procedure begins by extracting the *name* pattern and searching for the set of string tokens that fit this pattern across all DPEs using the *foldr* function. At the same time, the number of cumulative occurrences of this pattern is stored into the variable *occ*. If *occ* is greater than some θ threshold, the user is asked if the query is to proceed. If the response is yes or *occ* is under the threshold, the function *open-s* and the function name *fun* plus the list of string tokens *stoks* are used to construct a list of entity identifiers *eids* to which a reference is added to the frame memory.

Foldr is used for both searching operations so that the list is constructed in parallel. Each use of *foldr* would, of course, generate a separate graph so parallel execution could proceed. This is not shown explicitly in this example for the sake of brevity. However, the ability to do this is the essence of where functional programming and parallel processing techniques combine to allow the fine-grain parallelism that is required.

The ability to pass down generators and filters bound to the same variable, is captured in the next example procedure for *Restrict*. This function has a similar appearance to *string-map* but now has to cope with all types—not just strings—and closes the procedure by making an intersection of each set of entity identifiers created for each predicate in the filter. In this procedure *fun, op* and *val* are the function name, operator and value of each predicate and *ext* is the entity class extent used in the search. *Vtype* holds the type of the value *var* and, once again, *occ* is used to hold the cumulative total.

```
Restrict ri rj
      let ext == frame[FP+ri]
      let npreds == frame[FP+ri+1]
      for (i = 1, j = 1; i <= npreds; i++, j =+ 3)
            let fun == frame[FP+ri+j+1]
            let op == frame[FP+ri+j+2]
            let val == frame[FP+ri+j+3]
            let vtype == the type of val
            let ftok == get fun token from meta triples
            if (vtype == string)
                let (occ,vtok) ==
                    foldr (k rest) APPLY dpe k string-match val
                    (0, nil)
                    dpes
                in
                    if (occ > 0) AND NOT (ok=ask_usr(occ))
                        frame[rj] = abort
                    else
                        hd(vtok)
            else
               let vtok == transform(val)
               in
               let eids[i] == foldr (m rest) APPLY dpe m open-s fun vtok
                              nil
                              dpes
               in
                  frame[rj] = $\bigcap_{i=1}^{npreds}$ eids[i]
      add descriptor (FP, IP+1)
```

The *invert* procedure handles inverse function mappings of the form (inv_fun, op, value) and is similar to *restrict*. The values are mapped to tokens, asking the user if the query should continue where applicable, before assembling a set of entity identifiers as the result. Lastly in this section, we show the *select* function used to locate attributes for a given entity identifier or list of identifiers for printing purposes.

```
Select ri rj
      let ext == frame[FP+ri]
      let atts == frame[FP+ri+1]
      let eids == frame[FP+ri+2]
      let (rtoks,vals) == foldr
            (k rest) APPLY dpe k get-recs ext atts eids
            (nil,nil)
            dpes
      in
            frame[rj] = hd(rtoks,vals)
            add descriptor (FP, IP+1)
```

This procedure passes down the extent to be searched, a list of entity identifiers and a list of relation tokens that are used to retrieve the attribute values. The storage sub-system function *get_recs* accumulates the results to be displayed and in turn uses function *get_attributes* (discussed in chapter 5) to extract the required attributes. Using *foldr,* each disk in the array retrieves its list of relation-token/attribute-value pairs (*rtoks,vals*) to construct its list in parallel. These would then be used for any display and print options.

7.5.3 Graph analysis

A simple heuristic developed by Iannucci [IAN88] called the Method of Dependence Sets (MDS) is used to decide which graphs are worth reducing in parallel and which are more concisely performed sequentially. As a simple example consider the two ways of reducing the following graph (shown in prefix) `(* (+ x y) (- x y))`.

```
x:     r10 ← x                x:     r10 ← x
       fork T1                       jmp T
       jmp T2                 y:     r11 ← y
y:     r11 ← y                T:     join r9 b0
       fork T1                       add r12 r10 r11
       jmp T2                        sub r13 r10 r11
T1:    join r9 b0                    mul r14 r12 r13
       add r12 r10 r11
       jmp T3
T2:    join r9 b1
       sub r13 r11 r12
T3:    join r9 b2
       mul r14 r12 r13
```

With full parallelism   With partial parallelism

Note the offsets, *rn* above, have been chosen so as not to clash with control offsets etc. as described in section 7.5.4. With full parallelism, synchronisation occurs for inputs *x* and *y* and also for the *join* instructions. In the partially parallel version the only synchronisation is for values *x* and *y*. After that, all three arithmetic operations are performed in sequence. With around 50% less instructions in the partial version, it makes sense in this instance to use sequential execution and forgo the possibility of executing the arithmetic operations in parallel. The basic idea behind MDS is that parallelism is preserved between long-latency operations while sequential code is used for connected sub-graphs that do not include such operations.

We include MDS in our architecture because it is well understood, provably deadlock free and the latency-directed approach is appropriate for long-latency persistent parallel systems. Comparing it to other approaches is not part of this thesis. As with list comprehensions, the rationale is that a little pre-processing can prove advantageous in the final outcome. MDS analyses and partitions the DFGs into parallel and sequential 'blocks' of instructions. A simple example is shown below

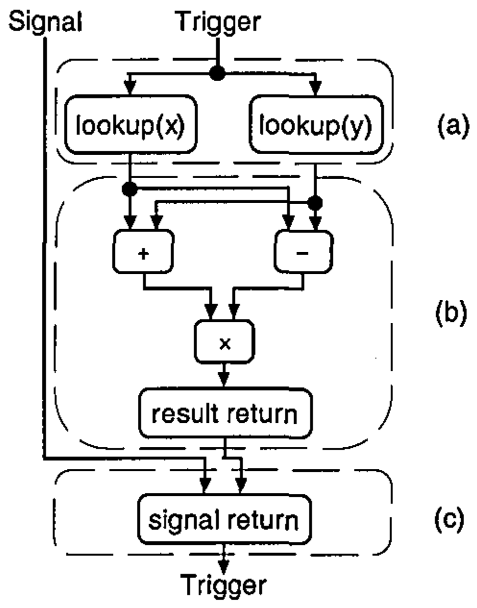


Figure 7.9. Methods of dependence sets.

In this graph the long latency sub-graph—partition (a)—is made parallel. This is because any operation that involves the *lookup* operator could mean access to the storage sub-system is required. The arithmetic sub-graph—partition (b)—is sequential, executing as + then – then *, as shown in the parallel code on the previous page. While partition (c) is again parallel. This is because (b) depends on (a)'s outputs and (c) depends on (b)'s output plus the initial signal token. The partitioning is performed on the graph in the body of each procedure definition—the instructions encapsulated by each lambda. Iannucci's algorithm for determining dependency sets is as follows:

1. topologically sort the instructions in each lambda block
2. uniquely name each long-latency output in the graph
3. for each instruction *i* in the block calculate the dependency set (DS).

$$DS(i) = \left(\bigcup_{j \in J} DS(j)\right) \cup \{o \mid o \in O \wedge long\_latency(o)\}$$

Where *J* is the set of instructions from which *i* receives input and *O* is the set of output arcs that connect instructions *J* to *i*.

7.5.4 Allocation of frame slots

Here the temporary storage implicit in the DFG's is mapped to frame slots that hold the synchronisation and values required. Each transaction has one *control* frame to hold details of the transaction, plus any number of *procedure* frames where one frame maps to each user-defined procedure—shown in Figure 7.8. All frames consist of lineally addressed memory. The control frame merely consists of a slot for frequently used constant 0, a slot for a self-pointer (self FP) and as many other dynamic slots as are required. Procedure frames are more complex and are shown and described below.

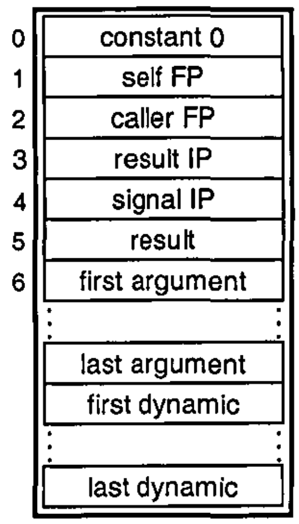


Figure 7.10. Procedure frame.

After storing constant 0, self FP, caller FP, result IP and signal IP, the *result* slot refers to the slot in the caller's frame where the result is held. Then there are slots for the procedure's arguments followed by dynamic slots. The algorithm for mapping graph storage to frame slots, traverses the graph and, for each instruction, slot bits for synchronisation, internal use and outgoing, value-carrying arcs are added. Static analysis determines when a slot is no longer in use and can therefore be considered for re-use quite safely.

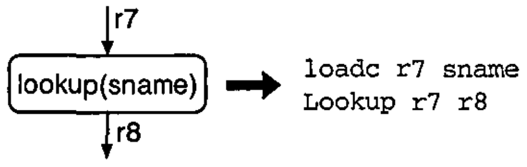
Because the execution of instructions may be unspecified, static allocation may not always be enough to guarantee the safe re-use of dynamic slots. For example, if a slot *ri* holds the result of a procedure application, it is only after all the following users of the result have consumed its value that it can be safely re-used; this may not be determinable at run-time. The rule used to determine safe re-use is summarised as follows. If there is only one consumer, slot *ri* can be safely marked for re-use. If all consumers reside in the same partition and are executed sequentially, the slot *ri* can be safely re-used after the last one has been fired with its value.

Two sets of free slots are maintained. One set *free-slots* contain those that are immediately available. The other set *pending* contain slots that will become free after the current level of graph instructions are complete. Free slots cannot simply be carried forward to the next node in the graph because of the above discussion on allocation. There has to be a one-node delay before a slot can be considered safe to re-use.

7.5.5 Mapping DFGs to parallel code

In this section we show how the DFG operators introduced earlier are mapped to the parallel code just discussed. Parallel code for the transaction consists of the expanded code of the body followed by frame de-allocation, using the `DeAllocate` instruction, and thread termination, using the `die` instruction. The die instruction ensures that when the partitions are appended, there can be no 'dropping through' the code of one section to the code of the next section. The final transaction code consists of a header, the frame size, base address, the parallel code and a static data area where strings are stored. It is not in the interests of efficiency to copy or even simply move strings around in memory, so the static data area is where they are kept.

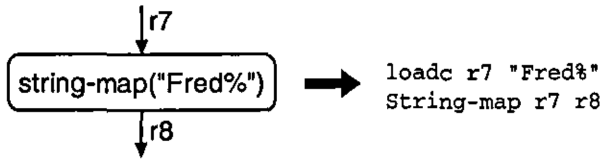
Constants are mapped with the *lookup* operator using the `loadc` instruction followed by the `Lookup` procedure.



```
loadc r7 sname
Lookup r7 r8
```

Figure 7.11. The *lookup* operator.

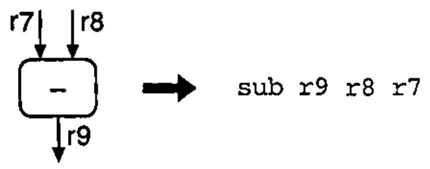
This simply places the constant value into an offset of frame memory for the `Lookup` procedure to handle the mapping to a token. Meta string constants would not be moved from the static data area; a pointer to them is passed into the offset of frame memory. For string patterns, the *string-map* operator translates to the procedure `String-map`, again after the patterns have been placed into frame slots beforehand.



```
loadc r7 "Fred%"
String-map r7 r8
```

Figure 7.12. The *string-map* operator.

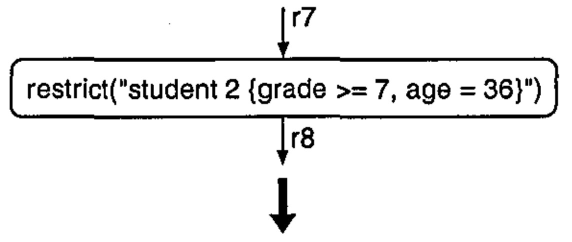
The standard primitive function operators easily map across to parallel code. For example, the subtract function is as follows



```
sub r9 r8 r7
```

Figure 7.13. The *subtract* operator.

Included in this group are our built-in functions that operate by passing down filters and inverses directly to the storage sub-system. The procedure *restrict* translates as follows



```
loadr r7 "student 2 {grade >= 7 age = 36}"
Restrict r7 r8
```

Figure 7.14. The *restrict* operator.

The procedures for *inverse* and *select* are similar. Finally, the basic idea behind the encapsulating lambda procedures is summarised. Recall that these are used for user-defined functions. Expansion of these functions recursively expands the body of procedure code. The procedure code is placed in the static data area for the overall transaction and a reference to it (the result) is placed in the result slot via `loadc`. Result-return and signal-return translate to the instructions: `start1 r2 r3 r5 r10` and `start0 r2 r4` respectively.

The general translation scheme for user applications is this. First, synchronisation is provided for procedure input (if required) and a new frame of the appropriate size is allocated. When the frame is available, threads in the procedure body receiving the trigger, signal and all arguments not requiring synchronisation are started. Arguments requiring synchronisation are started when both the argument value and new frame pointer are available. Lastly, threads are set up to dispatch the result and termination signal.

7.6 Performance improvements

As explained in the introduction to this chapter, performance gains are more speculative regarding the provision of a parallel implementation. For this reason, and because our dataflow approach is motivated by the AGNA system, we first give their improvements and then describe how our projected improvements are possible given our enhancements described in earlier chapters.

From the empirical results of AGNA using a 50,000-record database at 202 bytes per record, the following improvements apply [HEY91]. With a single processor and no index, improvements are as follows. Using low-level filtering—where the result list is constructed and filtered in the storage sub-system rather than on the heap—query evaluation times reduce from 148 seconds to 34 seconds (a 75% saving). There is a slight increase in additional time spent in the storage sub-system using low-level filtering—from 8 seconds to 12 seconds. However, the largest saving is in the reduced time taken to check the predicate—from 114 seconds down to 4 seconds. Applying indexes reduces response times considerably.

For a single processor using indexes, the initial search figure of 34 seconds comes down to 1.1 second. The AGNA system uses an extensive assortment of indexes and quotes a total transfer rate of 0.56 megabytes per second. Our architecture incorporates a data filter capable of a transfer rate of 12 megabytes per second on each processor coupled with a much coarser indexing structure. Their quoted figure of 1.1 second using a B-tree comes down to 0.84 of a second using a data filter and a full scan of the 50,000 records. However, the average number of records to be scanned could reduce to around half that—thus taking 0.42 of a second. In both cases time taken to construct the result list is negligible at 0.04 of a second.

When the same query was run using an INGRES database the execution time was 0.3 of a second. Therefore, potential response times using our approach would come somewhere between the B-tree-indexed, persistent object system (AGNA) and the commercial database product (INGRES). Furthermore, processing queries in parallel enhances performance rates still further.

Speedup is linear for non-indexed access, whereas for indexed records the optimum number of processing elements—called DPEs in our architecture—is frequently quoted as eight [CHA96]. However, we suggest using ten DPEs for the RAID system proposed and feel this is a reasonable compromise. Scale-up—where the database extent is increased in proportion to the number of processing elements—shows response times are constant where there is no index. Where an index is used there is a gradual increase in response time in relation to the number of machines used. This is due to the increase in communications overhead.

The above figures all assume an even distribution of record placement across the processing elements. We showed this is possible and indeed far more likely for large data sets by generating random numbers to simulate entity identifiers and checking the standard deviation was within acceptable parameters.

Further comparisons between our approach and INGRES, when parallel evaluation of queries is taken into consideration, show the following. Relational databases, such as INGRES, have weaknesses when used for graph traversal operations because of the 'flatness' of the relational data model. Furthermore, in languages such as INGRES, complex operations have to be embedded in a host language that gives rise to impedance mismatch problems and the difficulty of allowing the compiler to optimise expressions. Additionally, there is a lack of expressive power and semantics in relational languages, which is only now being addressed in standards such as SQL:1999. But, while these are still in their infancy, functional languages have used such features for many years.

Finally, the concept of using the RAID copy directly for inverse function steps in graph traversal operations means traversal times are reduced. From the empirical comparisons between AGNA and INGRES cited earlier, reverse traversals were between 1.9 and 2.6 times slower than forward traversals. This disparity is not applicable in our system.

7.7 Discussion

Following on from chapter 6, we have introduced further concepts in this chapter that also form part of our architecture. These are: transformations to a core sub-language; the use of optimisation techniques; the use of an abstract reduction machine; and translation of sub-expressions to parallel code using dataflow graphs. The aim is to produce fine-grained threads of code for parallel execution. These choices are now discussed.

7.7.1 Using a core sub-language

Using a core sub-language taken from the query language is a standard way of reducing a query language to a more manageable sub-set for the compiler to work with. The sub-language has eliminated from it all features of the query language that make programs easier to write, which at the same time add no new expressive power. The smaller set of instructions can then be used with the reduction machine to reduce expressions in a more controlled way.

Using a reduction machine also complements the functional paradigm more naturally. Moreover, if an expression can be optimised by passing down generator/filter combinations directly to the sub-system, the user need not be aware of this. So functions like `restrict` can be used safely as they are brought into use by the compiler, not the user.

7.7.2 Optimisations

The optimisations introduced in this chapter fall into three categories. Firstly, there are the traditional algorithmic- and implementation-based optimisations that have been around for some time. Secondly, there are the optimisations first introduced by AGNA that pass down selections and projections (generators and filters) to the storage level for evaluation using open lists. Lastly, there are our optimisations that pass down string and text functions to the storage level and those that improve the treatment of inverse functions.

The functions that operate on strings and texts are an enhancement of the AGNA work as they have the capacity to offer far more powerful searching operations to the user. Inverse functions are handled by adding an operator to the expression and passing this down too. For entity-entity traversal, the storage sub-system can then use the inverse triples and the operator to produce a set more quickly than by using a generator-filter combination and inferring inverses by software alone. Again, this is not obvious to the user.

7.7.3 Why use dataflow?

The central theme in this chapter has been the strategy of using a dataflow model supporting a MIMD parallel architecture to exploit the inherent parallelism in functional languages. Our model uses fine-grain parallelism with data-driven execution—both for computation and for long latency disk input/output. Earlier research into dataflow architectures has clearly shown this is an efficient way to mask the long latencies inherent in a parallel computer [MOR99] [NAJ99]. As a model of computation, dataflow has a long history. It has demonstrated its flexibility and efficiency in representing computation by the wide variety of areas in which it has been used but is particularly suited to the functional paradigm [JUL97].

7.7.4 Comparison with AGNA

AGNA was developed as a persistent object system using the functional paradigm and list comprehensions built around a lisp-like syntax. Coupled with a MIMD parallel processing model and dataflow graphs, AGNA showed how subordinating tasks to a lower level can vastly improve upon one of the most serious shortcomings of functional languages: performance.

However, in AGNA the user has to formulate queries in such a way as to make predicates for passing to the storage sub-system explicit to the compiler. We do not take this view, instead preferring the use of optimising techniques applied to queries supplied directly by the user in their raw form. This is easily achieved as it merely involves collecting together similar variables with their functions which are bound to the same entity class.

Examination of the schema information during the optimisation process identifies functions that range over the same domain thus enabling grouping of predicates to be done safely. One way to do this is to move sub-expressions around within the expression list that the user supplies in their query. This is the way many list optimisations are handled. Again, the strategy is that a little more time spent in the pre-processing stage can pay dividends in later query evaluation—especially when handling large data sets.

AGNA made extensive use of indexes on all object fields. This has obvious implications for update routines but was in keeping with the thinking behind relational languages that use similar strategies. Using a coarse index-sequential structure and an on-processor data filter, our architecture is more suited to set collation of triples and records. Moreover, because of the way strings are held—in look-up tables—searches for incomplete strings can be multi-cast to all disks in the array, in the same way as triple requests are handled.

7.7.5 Parallel Haskell

Included in the proposals for Haskell 98, is the ability to call 'C' operating systems functions directly using the primitive *ccall*. The syntax of this is: `ccall proc` e_1 `...` e_n. Here *proc* is the name of a 'C' procedure and e_1 ... e_n are the parameters to be passed to it. This is used in their monads system for input/output [PEY93].

The *ccall* operator is, in fact, a constructor—not a function—that allows stricter control of its use and reduces type-checking constraints. However, it is not clear if the use of this operator can be embedded in a nested expression of arbitrary complexity as our string manipulating functions permit. Moreover, as our string manipulating functions are part of the language, they can be called directly and do not require an explicit call through the operating system. An example of *ccall* in a monad for 'putting' a character to the screen translates to the expression:

```
putcIO a = ccall putchar a
```

7.8 Summary

This chapter has examined three crucial areas in our architecture: transformation, optimisation and translation. Transformation involves converting a user query in the model language into a sub-language to make the work of the compiler easier. This is shown together with examples. Optimisation can sometimes be performed on a given user query—particularly when the query involves using a list comprehension. Earlier work in this field is summarised as an introduction to our particular optimisations that involve inverse functions, text searching functions and selection functions.

We adopted a scheme that uses dataflow graphs (DFGs) to show the translation of queries into sections of code that can be executed in parallel. DFGs were chosen because they are well understood and complement functional languages nicely. Again, we highlighted the general concepts before introducing our enhancements in the areas of text searching, inverse functions and low-level filtering of extents where examples are given to show how the concepts are applied. A brief description of graph analysis—using method of dependence sets—shows it has been proven that comprehensive parallel processing of queries is not always advantageous. The process of mapping DFGs to parallel code is covered showing how our functions fit into the overall scheme.

Finally, the performance gains possible are described. In particular, how our approach compares with AGNA and INGRES databases, and how our timings come between these two systems. When taking into account parallel processing and redundancy, improvements are greater. Using our RAID scheme for inverse graph traversal operations removes the disparity of execution times between forward and reverse searches in AGNA.

Chapter 8 Database creation, population and maintenance

8.1 Introduction

Up to this point in this thesis we have discussed a database architecture based on the triple store for all data and identified areas where there are weaknesses due to the homogeneous nature of this approach. Our answer to this is a new architecture that combines the benefits of the functional data model with those of the relational data model in such a way that parallel processing of graph traversal operations is more easily accomplished. Part of this architecture involves storing lexemes separately from entity triples and attribute records so they can be searched more easily. In this chapter we show how the standard operations such as database creation, population and maintenance are achieved and discuss integrity and security issues.

Throughout the rest of this chapter we frequently refer to two running example databases. The North Yorks crime database, with 2,501 crimes used for training purposes, and a larger database with 500,000 crimes. This is done to show how our architecture scales-up with a realistically sized database for timings and space requirements. The schema and triple breakdown for the North Yorks crime database are shown in appendices A1 and A2. In section 8.4 we introduce another, smaller crime database that is used in examples. This is to show instances of all data types and the outline structure of data in records.

8.2 Creating a new database

Creating a database often involves an experimental and development stage before the final database schema and other meta data is agreed upon. It is only then that full population of the database can proceed. In this chapter we use the term population to mean adding bulk data to the database to put it in a position where it can be used effectively. However, as part of the database set-up procedure various parameters and constant values need agreeing before population can begin. These include page and buffer sizes, reserved integers for the undefined (@) and unknown (?) null values that are supported, the MIN and MAX values for each of the type domains, and the reserved integers to be used for default entity values, plus any other reserved values. We note that the range of built-in types will be extended to include the type *text*, introduced in chapter 4, and the type binary large object (BLOB). BLOB types are used for non-textual attributes such as fingerprints and photographs and are deemed essential to provide for the richer type structure required in our application domain. The token space for each type is drawn from the domain of 32 bits and allocated as shown in Table 8.1. (This means, of course, that a 64-bit architecture can easily be catered for too.)

type	ratio	range	type label
String	4	00000000 to 3FFFFFFF	0 0 X X
Text	2	40000000 to 5FFFFFFF	0 1 0 X
BLOB	2	60000000 to 7FFFFFFF	0 1 1 X
Non-lex	1	80000000 to 8FFFFFFF	1 0 0 0
System	1	90000000 to 9FFFFFFF	1 0 0 1
Integer	2	A0000000 to BFFFFFFF	1 0 1 X
Real	4	C0000000 to FFFFFFFF	1 1 X X

(Where X indicates "don't care".)

Table 8.1. New type labels.

The justification for the intervals is largely based on what is in place in the current system (see chapter 2 table 2.1). The only difference being that a sub-domain for internal strings is no longer needed so this has been merged with that for strings. Recall that strings are now held delimited to word level, so a larger domain will be required for them. The range above allows for 1 billion instances—more than adequate in our case. The sub-domains for *text* and BLOB types are just arbitrary: 500,000,000 should be adequate for each. The sub-domain for *text* is used for document identifiers and the sub-domain for type BLOB is used for BLOB identifiers.

In addition to the above, the following data structures are also required. The schema table held in main memory (introduced in chapter 5) and the string tables and lexical triples both held on disk (discussed in chapter 4). The meta data is held in triples, as is the case now and can easily be accommodated in main memory—membership triples excepted. The meta data for the North Yorks crime database, whose schema is shown in appendix A1, needed 6,170 triples (9 pages) for storage and this figure is unchanging (it does not increase in relation to the volume of instance data). The meta data to be stored includes the 'base_load' files that TriStarp uses to hold base types, constructors and string functions as well as the standard manipulation functions, such as *head, tail* and *map* used in user-level functions. The addition of meta data is handled with the functions introduced in chapter 5, which are: *insert_type_dec(dec), insert_confun_dec(dec), insert_nonlex_dec(dec), insert_pfun_dec(dec)* and *insert_sfun_dec(dec)*. The ordering is on the record identifiers shown in Table 5.1.

8.3 Populating a new database

Populating a new database is achieved using macros that can handle the bulk creation of membership triples and instance data in the form of entity-entity triples and entity-attribute records. In this section, we describe the handling of membership triples and outline the procedure for instance data. The treatment of instance data is then explained in later sections.

8.3.1 Inserting membership triples

The number of 'is-a' or membership triples broadly reflects the sum of the entity instances. In the North Yorks crime database there were approximately 10 times more membership triples than crime records. (26,500:2,500). Membership triples cannot be wholly accommodated in main memory, so need spreading across the disk array. After the schema information has been loaded as part of the database set up stage, there are unique identifiers for the various non-lexical types to be added—*employer, person,* etc.

In the North Yorks crime database there are 2,501 records and 8 non-lexical entity types. This combination generated 26,500 membership triples. With a 16k-page size and 24-byte triples, this means page capacity is 682 triples—thus necessitating around 40 pages. The larger database, with 500,000 records, might generate 5 million membership triples requiring around 7,400 pages—too many to fit into primary storage. These need spreading evenly across the DPEs. Membership triples are inserted with the interface function *insert_non-lex_def(def)*. The format a membership triple takes is as follows:

```
<id_bits, "is-a", "employee", $e, NULL, timestamp>
```

where "is-a" and "employee" are reserved integers from the system domain and $e represents the randomly generated, 28-bit identifier for this particular instance of an employee. The "id_bits" and "timestamp" fields were discussed in earlier chapters. The fifth field is unused. Because of object migration no semantic can be attached to the entity identifier in field four.

Membership triples are ordered on their 'is-a' tag and, within that, on the field three type. The data placement algorithm uses the system-generated entity identifier (which is guaranteed to be unique) to decide which DPE will hold the triple. If ten DPEs are used—as suggested in section 7.6—then the placement algorithm needs to inspect the least significant digit (base 10) of field four where the range of 0..9 will determine on which disk the triple will be placed. This will ensure all membership triples are spread across all participating disks in the array.

Using as an example the larger database, the figure for membership triples is as follows. If 5 million membership triples are to be stored across ten DPEs, there will be around 500,000 to each disk. At 24 bytes per triple, this represents around 12 MB of storage required per disk. With 5 million randomly generated identifiers used for (say) 10 entity types across 10 disks, the probability of a fairly even distribution is quite likely. In fact randomly generating 5 million integers and distributing them in this way gave a mean of 500,000 and a standard deviation of 508.

To search 12 MB of storage with a search accelerator takes around 1 second. With the membership triples ordered on type, a coarse index (held in memory) enables an indexed sequential search method to be used to reduce search times further.

8.3.2 Inserting instance data

Populating a new database with raw data is done with the aid of macros. The use of macros has already proved successful with builders of TriStarp databases and a full description is given in the TriStarp manual [DOT96]. After macro creation (by the DBA), first the attribute records and then the entity triples are loaded, linked and inserted into the homogeneous triple store. However, in our situation there are some notable differences that reflect the alternative approach to storage that we adopt. First, there are different arrangements for storing entity attributes and entity-to-entity links. Second, inverse copies have to be created as part of our redundancy scheme. Third, we use different data structures for lexemes to provide the mapping to entity identifiers.

So the additions to these data structures are an important part of populating a database. The next sections show how additions of the various forms of instance data are handled using examples where appropriate.

8.4 Attribute records

One of the crime databases used for training purposes that we examined had entities *person* and *report*. We use these in the following pages with embellishments to show examples of all types used. The schema is shown in Figure 8.1 below.

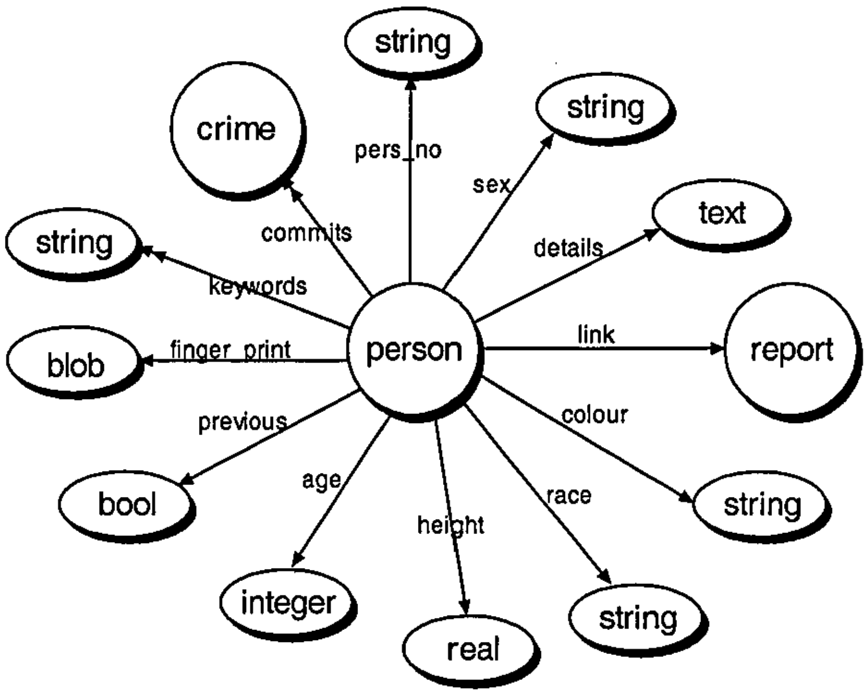


Figure 8.1. Raw data schema.

The raw data is held in records (usually one per line) grouped on a variable identifier for the entity class name—e.g., *per* for entity *person*. The data is entered using a typical higher-level graphical interface that controls the format of data entries. This is the standard way for data entry as used (for example) in INDEPOL [SOU97]. The attribute links in Figure 8.1 might be set out as follows:

```
per (pers_no, sex, race, colour, details,
                    finger_print, age, height, previous, keywords)
```

and an example of the mapping to instance data is shown in Table 8.2:

per		instance data
pers_no	→	"42015"
sex	→	"M"
race	→	?
colour	→	@
details	→	TEXT01 – link to file personal-42015.doc
finger_prints	→	BLOB01 – link to file finger-print-42015.pic
age	→	36
height	→	3.74
previous	→	TRUE
keywords	→	{"GBH","ABH"}

Table 8.2. Example of attribute data values.

The field descriptions are as follows:

field	description
"string"	quotes indicate string type (N.B. – a single character such as "M" is counted as a string.
?	reserved for unknown value
@	reserved for undefined value
TEXT01	references the document that follows
BLOB01	references the binary large object that follows
36	integer
3.74	real
TRUE	Boolean
{...,...,...}	used for list of attributes

Table 8.3. Description of record fields.

As mentioned above, it is important to stress that entering raw data into the system (not the database yet) is done using a strict format. This enables continuity to be maintained. Other constraints on data entry— such as how the text is entered and delimiters to be used—are not discussed here, suffice to say that there is a common 'look and feel' to all aspects of data entry. Entering an attribute record proceeds as follows. A new entity identifier—guaranteed to be unique—is generated from the sub-domain of non-lexical types (Table 8.1. refers). The attribute record is written to the DPE indicated by the last digit (base 10) in the entity identifier. In chapter 6, section 6.4.2, we introduced attribute records and the fields they comprise which have the following structure

```
<id_field,entity,{relations},{attributes},timestamp>
```

These are now discussed.

8.4.1 Id_field

This is a fixed length field holding the following information. The length of the record, the number of attributes in the record and one bit to indicate whether the record is 'live' or 'dead'—i.e. current or superseded.

8.4.2 Entity field

This is the fixed length, randomly allocated surrogate to be used as a guaranteed unique identifier for every instance of all entities.

8.4.3 Relations field

The {*relations*} fields—shown in braces to indicate the storage of a set—comprises ordered pairs of fixed-length relation identifiers (4 bytes) and offsets (4 bytes) into the record. So for the example in Figure 8.1 this field might contain

```
{<#pers_no,^offset>,<#sex,^offset>, ..., <#keywords,^offset>}
```

where each #x refers to the relation identifier and ^offset refers to its starting position later in the record.

8.4.4 Attributes

For {*attributes*} fields, when a record in raw data format is read the order of the fields is set out in the code of the macro created. Each field is comma-separated.

As such, the type of each attribute can be inferred from the schema data so that lexemes are mapped in the correct way for their respective type signature. This means attributes of type string (indicated by "......" around a value) require storing in the string table as space delimited words. For example, if a single-valued attribute had the value "bogus gas-man", there would be two entries in the string tables—one for "bogus" and another for "gas-man".

8.4.5 Timestamp

This is simply the 32-bit timestamp for time of insertion.

Unknown (?) and undefined (@) values are reserved 32-bit integers. This leaves the treatment of *text* types, *BLOB* types, multi-valued attributes and default attribute values to be discussed.

8.4.6 Text type

The way text documents are created is governed by the particular format for that class of document. Documents can have field identifiers, sections and sub-sections and, unlike straightforward string attributes, they are searchable. The reason they can be made searchable is that the type *text* can carry with it a whole range of functions to perform specific searches aimed at certain fields of a document. For example proximity searches—where it is required that word x be no further than y words apart from word z—are applicable to documents but not strings.

Moreover, document searches can seek words in particular sections of a document, such as in the abstract or not as the case may be. These searches cannot be done easily using the string triples alone. A discussion on this was presented in chapter 4.

When a database is populated each word in a document (stop words excepted) has to be added to the string tables. This is a similar process as happens with strings and, once done, the address of the document is added to the appropriate part of the attribute record. This is used if the document needs to be displayed. If it is desired, creating a set of document triples could provide an inverse mapping from document to entity (as with other classes of lexemes). Note that the type *text* is similar to the type *CLOB* (Character Large OBject) that is included in the proposals for what is now designated SQL:1999 [MEL02].

8.4.7 BLOB type

The entry in an attribute record that refers to a BLOB is merely the address of the object on disk. BLOB types are not searchable in the same way that text types are, but they might be in the future. Again, as with text types, the constraints for the type class will allow certain functions only to access BLOBs so the control is maintained at a type check level. We note that BLOB type is to be part of the SQL:1999 standard [MEL02].

8.4.8 Multi-valued attributes

Included in the attributes there may be some multi-valued relations (sometimes referred to as bulk data or 1:N relationships). We note from the foregoing discussion that these will be identifiable from the schema data (that checks the type signature) and the actual raw data where any multi-valued relations will be enclosed in braces {....} in comma-separated format. The layout for the attribute field of a record is

```
{ <attribute-length, value>, ... , <attribute-length, value> }
```

with examples (where the attribute-length value refers to bytes used)

```
{<5,"42015">, <1,"M">, <4,?>, <4,@>, <4,^TEXT01>, <4,^BLOB01>, <4,36>,
<4,3.74>, <4,TRUE>, <{<3,"GBH">, <3,"ABH">}>, <4,timestamp>}        (1)
```

8.4.9 Default values and intensional values

Recall from chapter 5 section 5.4.6 that default values for an entity attribute (lexical or non-lexical) are possible, as are intensional definitions. These are a fundamental part of the data model and must not be compromised. Examples of these are

```
  age (x:person) <= 30;       /* default function definition */
  age Fred <= age Mary;       /* intensional function definition */
```

There can be only one default function definition for each function in the database and there will be very few intensional function definitions. The schema table identifies which functions have intensional definitions and they are held in main memory as part of the schema data. In the attribute records (or entity triples), a reserved integer—EXP—associates the particular attribute with an intensional definition which is then looked up.

8.5 Storage requirements and placement on disk

Storage requirements and the placement of data on disk, is now considered for the following data types: attributes, string tables and lexical triples, entity triples and documents.

8.5.1 Attributes

Once an attribute record has been prepared and all the relevant entity mappings have been created in the lexical data structures, the record is written to disk. As with other data, the last digit (base 10) of the entity identifier determines on which disk the record should be placed.

Following the discussion on attribute layout, the total storage needs for attributes are as follows. From the above example (1) the requirement for the *person* record would be 56 bytes-per-record—not forgetting a byte for each attribute to hold the length. To this must be added the id_field, the entity identifier, the set of relation-offset pairs and the time the record was entered. The general rule for the storage requirement for attribute records for a database is summarised by the equation

$$\sum_{i=1}^{C}\left(\sum_{j=1}^{I} s_j + e_j + n_j(r_j + o_j) + t_j + (\sum_{k=1}^{n} a_{jk})\right)_i$$

where C = number of entity classes, I = instances of each class then, for each instance: n = number of attributes, s = id_field size, e = entity identifier field size, r = relation field size, o = offset field size, a = attribute size and t = timestamp field size. However, in the North Yorks crime database the above equation is not so easy to implement. Unfortunately, we do not have access to the raw data: it is still considered sensitive and thus classified. However, using the triple store, it is possible to extrapolate the data and this is shown in appendix A2. Some of this is reproduced in Figure 8.2 and used in the following sections which show the storage requirements using our new architecture and data structures.

The information in Figure 8.2 comprises four triple types. The *membership triples* map easily to our new architecture and are not the main focus of discussion here. The *entity →entity triples* are likewise straightforward to map across and are discussed in the later section 8.5.4. The other two triples types, showing the breakdown of *string triples* and *non-string triples,* are required in several data structures—namely attribute records, string tables, lexical triples and documents.

In all cases, the format *a →b* indicates a mapping from entity *a* to attribute *b*—where *b* could be another entity or a lexical value etc.

Numbers of membership triples for each entity type									
crm	2,501	itm	8,598	pit	5,002				
soc	2,501	gtn	2,501	mop	2,501				
day	2.770	bet	147					Total	26,521
Numbers of entity-entity triples (A → B) – relation name omitted									
gtn	→	itm	8,598	crm	→	gtn	2,501		
crm	→	mop	2,501	crm	→	soc	2,501		
crm	→	pit	5,002	soc	→	bet	2,501		
pit	→	day	5,002					Total	28,606
Numbers of string triples – including internal string triples									
crm	→	tag*	2,501	itm	→	cat*	8,700		
"	→	cir	69,000	"	→	des	118,000		
"		+	▨ 2,501	"		+	▨ 8.598		
"	→	scp	314,000	mop	→	hfe*	2,700		
"		+	▨ 2,501	"	→	mud*	4,400		
"	→	cno*	5,002	"	→	poe*	9,800		
"	→	oir*	2,501	soc	→	pat*	6,800		
"	→	ofn*	4,400	"	→	pcd*	2,501		
gtn	→	prt*	2,501	"	→	add	14,000		
"	→	gct*	6,700	"		+	▨ 2,501		
bet	→	bcd*	147	day	→	dywk*	2,700		
								Total	≈ 592,000
The remaining non-string triples									
crm	→	hoc	2,501	itm	→	val	8,598		
gtn	→	ant	2,501	mop	→	glv	2,501		
"	→	car	2,501	"	→	fgt	2,501		
"	→	cyc	2,501	"	→	fcn	2,501		
"	→	cmp	2,501	"	→	icd	2,501		
day	→	dno	2,770	bet	→	pph	147		
"	→	mno	2,770	"	→	osb	294		
"	→	yno	2,770	soc	→	irl	2,501		
"	→	dnwy	2,770	"	→	gqd	2,501		
								Total	47,630

Figure 8.2. North Yorks crime database – Triple allocation (part of).

In the section for string triples there are four (shaded ▨) that have an extra 2,501 (or 8,598) triples included in the figures. The string totals for these four are sufficiently large to require that each of the entity classes will need the string identifier triples as well as the totals for the (now broken down) string triples.

Examination of the data in appendix A2 shows there are over 639,000 lexical triples of which some 592,000 refer to string attributes. Comparing the number of triples required in this database, to the size of attribute records held in our structure is difficult to do. From the breakdown of triples it is reasonable to assume that the *scp* (definitely) and *cir* (probably) attributes would now be held as documents of type *text*. Whereas *add* and *des* would most likely stay as strings. This is because a sample inspection of *add* and *des* attributes showed strings are spread more evenly across their triples.

The 47,630 non-string triples all hold a value that fits into 4 bytes. To this must be added the 8 bytes for the relation-offset pairs and 1 byte for the attribute length per string. The large attributes that relate to *scp* and *cir*—now treated as text documents—have 383,000 triples between them (314,000 + 69,000). The count of words in the *scp* attribute gives a total of 315,000—very close to the triple total of 314,000. This is not surprising as each triple holds six characters of a string and the average length of a word in the English language is around six characters.

Applying this to the two text attributes gives 383,000 words or around 2.3 million characters to be stored which, at 1 byte each, equates to 2.3 MB for the document storage. In each record there will be 4 bytes for the link field, 8 bytes for the relation-offset pairs and 1 byte for the length (total 13 bytes). As there are 2,501 for each of the two text types, this translates to 5,002 documents sharing the 383,000 triples.

The remaining string triples total of 209,000 averages across 39,000 instances. This figure is arrived at by adding the entity identifier count for the string triples minus the text triples—the relevant attributes are marked (*) in Figure 8.2. This gives around 6 words (bytes) per record (209,000/39,000) to which must be added the 9 bytes above. Putting all this information together and scaling up for the larger database example, the storage requirements for attribute records are as are as follows.

non-string triples	47,630 x (4 + 9)	619,190
string triples (exc. documents)	39,000 x (6 + 9)	585,000
document values	5,002 x 13	65,026
actual documents	383,000 x 6	2.3 MB
record_length(2) + id_field (1) + entity surrogate (4) + second of entry (4) = 11 x 26,521		291,731
	TOTAL	3.9 MB
	x 200 for larger database =	773 MB

Table 8.4. Attribute record storage requirements (in bytes).

With an allocation of 10 DPEs, the above data is stored at around 77 MB per disk. In addition to the lexical values held in the attribute records, the requirements for the string tables and lexical triples have to be calculated.

8.5.2 String tables and lexical triples

The duplication of strings might at first seem wasteful. But, this is mitigated by the benefits of greater search opportunities (discussed in chapter 4), together with the redundancy value of duplicating the string table in reverse order. Moreover, because of the nature of our application domain (investigative systems) there is likely to be a recurring vocabulary in strings and texts following the initial loading of a few records.

By first eliminating 'stop words' the dictionary of the domain quickly becomes clear; an inspection of the North Yorks crime database confirmed that there is much duplication of short strings across records, and of individual words in long strings. In the current system, using the triple store, the first occurrence of a string generates a set of triples to hold the full string. Thereafter subsequent instances of the same string merely result in one triple being added to the store. In our case each new word added requires a new row in both string tables; repeats of a word only require the *occurrences* field to be incremented.

The size of the string tables is a function of the number of unique values to be stored. A standard dictionary might have O(68,000) separate references that we use as an example case in this section. The composition of a table row is as follows

word	token	length	occurrences
averages (say) 7 bytes	4 bytes	1 byte	2 bytes

so a reasonable estimate of the size might be 68,000 x 14 bytes = around 1 MB of storage required, although this is highly variable. This table is duplicated under the RAID architecture where the second copy is in reverse string order. The storage requirements for the lexical triples are more complex to calculate. In our example for the North Yorks crime database, there are 36 lexical attributes of which 18 are for non-string types. For these attributes each value is capable of being held in four bytes, so the triple total of 47,630 is the same.

For the 18 string attributes, the two large, text attributes for *cir* and *scp* are discussed below. The other 16 attributes are either one word attributes or represent multi-valued attributes. Even the attribute total for *des* (118,000) is not excessive bearing in mind the 8,598 instances of *itm* in the database. Moreover, examination of the *des* attributes showed there is very little word duplication. This translates into a triple-for-triple measurement. Therefore there would be around 209,000 string triples for these 16 attributes.

8.5.3 Text attributes

For the remaining two text attributes (*scp* and *cir* with 314,000 and 69,000 triples respectively) simply mapping their total triples of 383,000 to words is not accurate enough. It does not take into consideration word duplication that can occur in the same document. To reach a figure for the *scp* attribute we counted the number of words that re-occurred in the first 10 records. These first 10 records contain 1,113 separate, space-delimited words of which only 631 are unique, representing 56%. If this is applied to all the words that comprise the two text attributes—and this does not seem an unreasonable assumption—their combined word total of 383,000 @ 56% equates to around 215,000 unique words that require storage as string triples.

We note that there is also much duplication of strings across records, so the 215,000 unique words correspond to the number of string triples needed but *not* rows in the string tables. This is because every occurrence of a unique word in every attribute requires an entry in the string triples data structure. Putting the foregoing figures together we can arrive at a lexical triple store requirement as shown in Table 8.5 where a scaling up factor of 200 gives similar figures for the larger database of 500,000 records.

number of records	string triples	non-string triples	text attributes	totals ≈
2,501	209,000	47,630	215,000	471,630
500,000	41,800,000	9,500,000	43,000,000	94,300,000

Table 8.5. Lexical triple store requirements.

A summary of the storage requirements for all structures is discussed in section 8.5.6. Next we consider the addition and storage of entity triples.

8.5.4 Entity triples

In the current system the raw data is inserted into the database in the following order. First the lexical attributes are loaded using predefined macros that match the ordering of the raw data, then the entity triples (or links) are created using a unique key from the attributes. Again, this is done using macros and local variables to obtain the inverse (A→E) of a function (E→A) that gives the entity identifier to be used in the link. Using our example from Figure 8.1, we could imagine a link called *link* from *person* → *report* where we want to connect a person with *pers_no* "123" to a report with *rep_no* "789". (In this case the linking attributes are guaranteed to be unique and, as such, act like the primary keys in a relational database.) To do this, inverse functions are used on the two attributes to obtain the required *person* and *report* identifiers and the link is then set up. Instructions to do this as part of the macro are:

```
$p == head inv_pers_no "123";
$r == head inv_rep_no "789";
link $p <= include $r;
```

As inverse functions are not physically stored in the current system, the above sequence can involve considerable effort in collecting the required triples to map to the inverse. The result is always a list—hence the need to use *head* to extract the desired element. With our data structures we can use the string table to ascertain the entity identifier—given the lexeme and relation—without an exhaustive search of the records themselves.

Once the two global variables—$p and $r—hold the two entity identifiers for *person* and *report*, the triple can be written to disk. In the case of the above example, the following triple is constructed (where all fields are of fixed length).

```
<id_field, link, person, report, NULL, timestamp>
```

This is stored on the disk that shares the last digit of the *person* identifier. In the case of multi-valued *entity* → *entity* relationships, for example, the mapping *person* → *crime* via relation *commits* in Figure 8.1, the process is as follows. First each given key attribute value for the range (*crime*) is used to generate a range of entity identifiers, i.e. a set of *crime* identifiers in our case. Then the domain identifier *person* is obtained, as described above for inverse functions, before being mapped to each *crime* identifier in the set. This results in the following triple assignment—the existence of entity identifiers $p for *person* and $c1, $c2 and $c3 for *crime* is assumed.

```
<id_field, relation, $p, $c1, NULL, timestamp>
<id_field, relation, $p, $c2, NULL, timestamp>
<id_field, relation, $p, $c3, NULL, timestamp>
```

The storage on disk is such that triples are grouped into entity classes first and then ordered on relation field, domain field and (if needed) range field. Storage requirements are as follows. From the triple summary in Figure 8.2, we note that the North Yorks crime database has seven *entity* → *entity* relations that use 28,606 triples which, at 24 bytes per triple, amounts to a storage requirement of 686,000 bytes. For the larger database of 500,000 records there would be 5,721,200 triples needing 137,308,800 bytes of storage. When these are spread across 10 DPEs and duplicated for RAID and inverse function use, there are around 27 MB on each DPE.

8.5.5 Documents

Finally, in this section, we must consider the additional storage requirements for the field delimiters in the actual documents themselves. Recall that the 383,000 triples equate to 2.3 million characters (bytes) at 6 bytes per triple. Also recall that the number of triples relates closely to the number of words. So, with 383,000 words split into an educated guess of 14 words per sentence, there would be around 27,300 sentences. Each of these needs a start and stop byte giving 55,000 bytes of storage.

To this must be added some bytes for section and paragraph delimiters although there will be fewer of these. It's not easy to arrive at an accurate figure for the number of sections or paragraphs, as each document will differ greatly. However, using the statistics from this chapter for example, we can estimate that there could be around 1,000 sections and 4,000 paragraphs in the documents that total 383,000 words. This would result in another 5,000 start and stop bytes needed totalling 10,000 bytes.

The final analysis of individual documents could vary greatly but, using our assumptions, the revised amount of space required for the documents is now shown below.

database size	document word size	bytes required	plus sentence delimiters	plus section and paragraph delimiters	total storage requirement in bytes
2,500	383,000	2,298,000	55,000	10,000	2.36 MB
500,000	76,600,000	459,600,000	11,000,000	2,000,000	472 MB

Table 8.6. Total document storage requirements.

8.5.6 Total storage requirements

To summarise this section, we give figures for typical storage requirements with the running examples used. In some areas the figures are fairly arbitrary but give an indication of how the architecture scales up. The figures also include an amount for a temporal index: this is discussed in the next section.

	2,500 record crime database		500,000 record database	
	triples or records	storage required (in bytes)	triples or records	storage required (in bytes)
schema triples	6,170	148,000	6,170	148,000
membership triples	26,521	636,000	5,304,000	120,000,000
entity triples	28,606	2 x 686,000	5,720,000	2 x 137,300,000
attribute records	26,521	1,561,000[†]	5,304,000	312,200,000
documents	-	2,360,000	-	472,000,000
string tables	-	2 x 1,000,000	-	2 x 1,000,000
lexical triples	471,000	5,652,000	94,300,000	1,131,600,000
temporal index	-	negligible	-	880,000,000
TOTALS		around 13.7 MB		around 3.2 GB

Table 8.7. Total storage requirements.

For a comparison with the current storage requirements of the Birkbeck Triple Machine (BTM) the following is noted. Of the triples shown in appendix A2, the crime pattern analysis triples are not applicable; they were added to the software for a specific purpose and are not included in our calculations in this chapter. The remaining 710,095 triples are packed 382 triples-per-page with a page size of 16,384 bytes. Comparison with the figure of 13.7 MB in Table 8.7, however, has to be made assuming a full page capacity of 682 triples (as we have done above). Therefore a fairer measure of the storage requirement for the BTM is as follows.

$$\frac{710{,}095 \times 16{,}384}{682} \approx 17\text{MB}$$

The 13.7 MB above provides a space saving of 3.3 MB or ≈ 19% on the BTM.

In our system, the placement of data on to a disk in the array is shown in Figure 8.3 where each section has a pool area for additions.

[†] Taken from Table 8.4 minus the actual documents themselves.

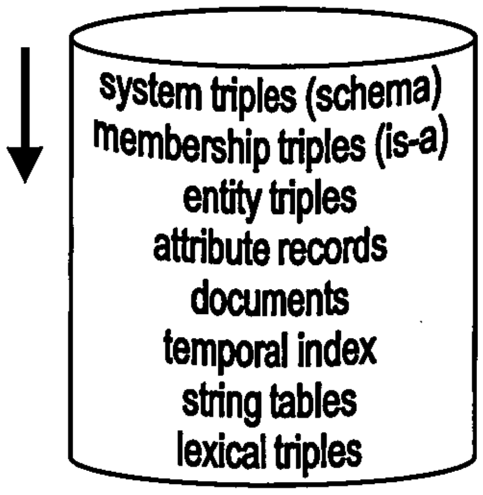


Figure 8.3. Storage architecture.

8.6 Indexing

One of the features of this architecture is that indexing is kept simple and at a coarse level. This complements the use of a search engine. There is little to be gained—indeed time can be added—in maintaining a complex indexing structure to give a more precise entry point into the disk arrays [MAL79]. Often such complex indexes cannot fully reside in main memory and overheads can accumulate as sectors are retrieved to follow a potentially lengthy trail of pointers to the data. Maintaining the integrity of complex indexes is also costly. However, index provision is an essential part of a storage system, so the proposals for our architecture are set out in the following sections.

8.6.1 Schema triples

There are only 6,000 or so of these which, at 24 bytes each means storage of 148,000 bytes. This is sufficiently small and can be accommodated wholly into main memory occupying around 12 pages using a 75% occupancy rate. Indexing is therefore not an issue.

8.6.2 Membership triples

With around 12 MB per disk this gives a complete search time of around 1.2 seconds using sequential scan (n = 12 MB and search rate = 10 MB/sec). If each disk had a local index for entity class boundaries within its allocation of membership triples, this would give an indexed-sequential entry point for each class of entity on each disk to be used if required. This could bring scan rates down to an average of .6 of a second.

8.6.3 Entity triples

At 24 bytes per triple and duplicated in inverse function order, these require around 275 MB of disk space spread over 10 disks. Again, a simple index on entity class boundaries provides an indexed sequential method of access to reduce the 27+ MB to be searched on each disk. Note that each disk needs two indexes—one for the normal function mapping and one for the inverse that doubles as a RAID copy.

8.6.4 Attribute records and documents

After the lexical triples and temporal index structure, this set represents the largest to be stored on the disk array. Inspection of the storage requirements for the attribute records and documents in Table 8.4 reveals the following. Of the 3.9 MB total storage for the North Yorks crime database, around 2.3 MB refers to documents pointed to from their respective record via a document identifier. Scaling these figures up by 200 means that, for the larger database, there is around 300 MB relating to attribute records and around 470 MB relating to the documents themselves. Therefore searching the index for attribute records is confined to the 300 MB held in class order on each disk at 30 MB per disk and, within that, in relation and entity identifier order. As before, using indexed sequential access and a search accelerator, the required records can be identified on each disk in parallel for return to the controlling processor.

The documents themselves are searchable and have structure (sub-sections) within them and are indexed indirectly via string identifiers and entity identifiers. Documents also have their own set of search functions applicable to *text* type attributes only—as described in chapter 4. The start and stop section markers can be used as a (very) coarse index to specific points within a document from where a sequential scan can begin.

8.6.5 String tables

The strings are duplicated in reverse order (the RAID copy) and, although the size of these tables is very dependent on the particular vocabulary for the application, the indexing will always be fairly coarse. In our earlier examples we quoted 68,000 unique strings giving around 1 MB of storage for each table. However, this figure could vary considerably.

In our experimental programs, where we searched through 5.5 million strings, we used an index based on the 26 letters of the alphabet. This proved sufficient to locate a starting point for sequential scanning of a sub-set of the strings. Following the collation of a set of string tokens, the next stage is to map these to entity identifiers via the lexical triples. The size of these is considerably larger and is discussed next.

8.6.6 Lexical triples

As there are in excess of 1 GB of lexical triples to store, the index for these triples represents one of the largest to be maintained in the system. Recall that the lexical triples are ordered on each disk as follows:

1. sort into entity class order
2. within that sort into relation order
3. within that sort into lexical token order

4. within that sort into entity identifier order.

Table 4.10 in chapter 4 gives a small example of the structure for storing string triples. Following our earlier example of a 500,000 record database, the 94.3 million lexical triples require indexing as follows.

Non-string triples

From Table 8.5 there are 9.5 million of these. They provide the mapping from non-lexical types—such as integer and Boolean—to entity identifiers. Ordered as above, they are spread across the disk array by a hash value taken from the lexical value. For example, a triple with the pattern:

$<$`relation,` 23135_{10}, `entity_id`$>$

would be placed on disk 5 because 5 is the last digit of the lexical token. On each disk indexed sequential access follows a coarse index on the relation.

String and text triples

These form the bulk of the lexical triples—over 84 million in our example (41.8 million string + 43 million text)—and are ordered as above. As they are created, loaded and sorted at the time the database is populated, a standard way to index such large numbers is via a B-tree. However, if we wish to maintain consistency with a coarse granularity indexing structure, an indexed sequential method is used on each disk to search around 8.4 million triples which, at 12 bytes-per-triple, gives around 100 MB to search at around 10 MB on each disk.

8.6.7 Temporal indexing

A feature recognised as missing from the original TriStarp proposals, was the ability to make searches of the triples in historical context. The tag field used to store the date of insertion was not easily searchable as it was not specified as a secondary key. This situation can be remedied by building a secondary index on the timestamp field. This is done on database set-up and updated on database re-organisation. As the ordering of these triples is a sequential set, an appropriate data structure is a B^+-tree [KNU73]. The salient features of which are:

- an integer M controls the maximum children allowable for each node
- all nodes, except the root node and terminal nodes, have at least M/2 children and not more than M children
- all terminal nodes are at the same level, and thus the same distance from the root node
- a non-terminal node with k children contains k – 1 keys
- terminal nodes represent the sequence set of the data file.

In our case we are using the structure in a static way, so the level of bushiness, M, can be determined by how much information can be accommodated on a page in memory. If a page in memory is taken as 16,384 bytes, then the following needs to be held in it. A number of index keys—in this case representing the 4-byte timestamp fields—and a set of pointers to children (other pages in the tree) that are accessible from this node. We have chosen 4 bytes for the tree pointers (2^{32} giving ample range) although 3 bytes could equally have been used.

The timestamp field need not be held in its entirety as the key. Some of the high-order bits in the timestamp field are identical—and therefore could be factored out of the index key. However, if we were to use a 10-year slot as the critical time slice, we would still require 29 bits in the index key; this number of bits is needed to store 315 million+ seconds that appear in 10 years. It is just as easy to store the full 32 bits of the timestamp field as the index key and therefore keep the field modulo 8.

The terminal node level does not need tree pointers but instead needs record/triple addresses that are also 4 bytes long. For the non-terminal nodes, there must be one more tree pointer than there are index keys. So, if the tree pointer is 4 bytes long, this means a 16 Kbyte page for non-terminal nodes can best accommodate 2,047 index keys and 2,048 tree pointers (2,047 x 4) + (2,048 x 4) = 16,380 bytes. There are then 4 bytes free in which to store the current number of index records in the page. M is therefore 2,048. From Table 8.7, the depth of the B$^+$-tree for 5.7 million E→E triples, 5.3 million E→A records, 5.3 million 'is-a' triples and 94.3 million lexical triples is: $\lceil \log_{2048} 110{,}000{,}000 \rceil = 3$.

To accommodate all of the non-terminal index nodes in memory for 110 million triples and records could require 2048^2 or 4 million+ pages. This is looking at the maximum pages required. If the root level (1 page) and the second level (2,048 pages) were held in memory, 2,049 pages (33.5 MB) are required. Then there is one disk access needed to get the terminal node information from the last level of the non-terminal index nodes before the terminal node level itself can be accessed as a sequential set. Finally the triples/records themselves need retrieving. However, this is a rather expensive approach as shown in the following table.

level	pages	contents	storage needs
1	1	2,048 keys & pointers	16 KB in memory
2	2,048	4+ million keys & pointers	33 MB on disks
3	4+ million	max 8 billion keys & pointers	? on disks
then retrieve triples/records from disk			

Table 8.8. Storage requirements for temporal index.

A better scheme takes the storage requirements from the bottom up. For maximum efficiency, we need to determine the total storage needed for the terminal node pages and work up allocating non-terminal node pages as necessary. The terminal node pages require 110 million 4-byte keys and 4-byte triple/record pointers plus a pointer to the next page in the sequence.

index keys (2,047 x 4) =	8,188 bytes
triple/record pointers (2,047 x 4) =	8,188 bytes
sequence pointer	8 bytes
	16,384 bytes

So 110,000,000/2047 gives 53,738 pages to hold the terminal indexing information. At 16 Kbytes per page, this gives a storage requirement of 880 MB to be held on disk. The higher levels can then be held in main memory as shown in the following table.

level	pages	contents	storage needs
1	1	27 keys & pointers	28 pages and 460 Kbytes in memory
2	27	53,738 keys & pointers	
3	53,738	110 million keys & pointers	880 MB on disks
then retrieve triples/records from disks			

Table 8.9. Optimised temporal index requirements.

This shows the allocation of pages optimised for placing the non-terminal levels into memory. Now just one disk access is needed to get the terminal index node information.

The terminal node level of the index can be held in a different data structure, as it is a sequential set. The 880 MB of storage required is spread evenly across the disk array—giving around 88 MB to search on each disk in a 10-disk array. In order to involve each disk in a temporal search and spread the load evenly, the same placement algorithm used for entity triples can be applied here. In other words, the least significant digit (base 10) of the index key is used to determine which disk a 'key/disk address pointer' combination goes on to. Searches now proceed by multiplexing the range of seconds and collating the results. On each disk, an indexed-sequential search is again used.

Note that the structure in Table 8.9 has flexibility built into it. Storage at the first and second level could be raised to 1 MB—for example—using around 62 pages. These pages would provide for at least 129,000 terminal node level pages which, in turn, could hold up to 264 MB of 'key/disk pointer' pairs.

8.7 Adding and deleting data

One of the benefits of functional programming languages is referential transparency, which means the value of any expression is immutable. Therefore functional programming languages enjoy ease of reasoning, freedom from detailed execution order and, freedom from side effects. However, the price to pay for these advantages is that assignment, or updates-in-place, are not allowed. Because updates in a *purely* referentially transparent database are so difficult, many languages—often referred to as *impure* languages—compromise and use assignment. The consequences of this can be severe.

A detailed evaluation of different update methods is beyond the scope of this thesis. However, some examples of different approaches are as follows. McNally *et al* [MCN90] use response/request streams. These incorporate lazily evaluated lists but are hard to write and understand. The scoped referential transparency scheme of Meredith and King [MER98] permits updates by functions with side effects. To counteract these side effects, an *effects* checker is used. In parallel Haskell [ARG87] a tree structure is used to underpin the database. Each time an update occurs, only the nodes in the path from the root to the new value are replicated.

FDL was developed as a *pure* language where updates can only occur at the 'top level' by using **let .. in** constructs. This means that it is not possible to perform a group of assertions atomically in a concurrent program. The way this is handled is by marking the triple to be amended as deleted and then inserting a 'new' triple with the amended value. The old triple is then archived at a later date and replaced by the new one. In our case, we are now dealing with records and triples but there is still scope for the 'deletion and insertion' method as above plus a temporal dimension as previously described.

Recall from chapter 4 that in our application domain databases are primarily used for browsing and searching. Data entry tends to be done in batch entry mode with the database being updated and optimised off line. In chapter 3 we identified that only 5% of the UK Inland Revenue files required alteration (on a daily basis) and that this could easily be done overnight. From the earlier discussion in this chapter, it is clear that our treatment of adding bulk data is that this is done statically. This means it can be optimised for such tasks as indexing and searching. However, any working system must cater for the dynamic dimension with new data being available to browsers immediately. So, additions and deletions of the various types of data are as follows.

8.7.1 Membership triples

As these are spread across the disk array ordered on their last digit, new triples can be directed to the appropriate disk for addition to a pool area at the end of the sub-section of that disk. A re-ordering of the triples can be done when a suitable threshold is reached on each disk in the array independent of other disks. Examination of the id_field of any triple identifies whether the triple is 'current' or 'deleted', although deleted triples are not removed at the time of deletion. The new triple added to the pool area supersedes the old one and is used in any operations that involve its entity class. In our example database, 5%

of 5,300,000 triples gives an average of around 265,000 updates each day which, at 24 bytes per triple, equates to 6.3 MB.

8.7.2 Attribute records and documents

Attribute records are updated in the same way—by marking a record 'deleted' in the id_field and adding another to the pool area in the sub-section of the disk reserved for attribute records. The only difference being that the average length of an attribute record is 150 bytes compared to the 24 or 12 bytes for other triples. 5% of 5,300,000 attribute records would give an average daily update of 265,000 records—around 40 MB of data to be updated. Document additions/deletions are treated similarly. However, when a document is deleted, the words in it also need deleting from the string triples and the string tables *occurrences* field is reduced accordingly.

8.7.3 Entity triples

With 5.7 million that need duplicating, there will be a total of around 11 million+ triples, where perhaps 500,000 need updating each day. Again, the id_field will identify the status of a triple *in situ*. Furthermore, for this type of data—where there is always a fixed length—better placement of data can be achieved.

8.7.4 Lexical triples and string tables

Each time a new word is added a check is made to see if the word already exists in the string tables. If it does, the only action is to increment the *occurrences* field and then insert the required lexical triples. If the word does not already exist, a new record entry is created in the string tables with a new and unique string identifier allocated for the new word. Any lexical triples are then added as necessary. Deletion of a word from the string tables can only be permitted when there are no other uses of that word in the database. Provision can also be made to archive unused words.

8.7.5 Schema data

From the information obtained from the North Yorks crime database, schema triples amount to only 6,170 representing 148 Kbytes. This figure is largely invariant and can be accommodated quite easily in memory. However, any major changes to the database schema can involve altering large amounts of instance data (records and triples) and membership triples—although, because semantic-free identifiers are used, there should be no need to alter lexical triples.

For example, object migration might mean an entity, *E*, is split into two entities, *E1* and *E2*, where some of the attributes are carried over to *E1* and others to *E2*. This is shown in the next simple example where we assume two instances of entity *person* whose identifiers are 1234 and 5678 respectively.

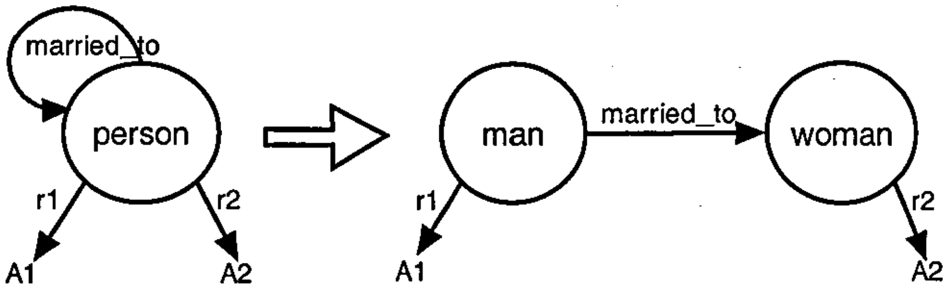


Figure 8.4. Schema changes.

The triples accompanying the *person* entity and how they need changing to reflect the new entities *man* and *woman* are shown below followed by an explanation of what data is affected.

	With *person* entity	With *man/woman* entities
1	<is-a, person, 1234>	<is-a, man, 1234>
2	<is-a, person, 5678>	<is-a, woman, 5678>
3	<married_to, person, person>	<married_to, man, woman>
4	<1234, (r1, A1), (r2, @), t>	<1234, (r1, A1), t>
5	<5678, (r1, @), (r2, A2), t>	<5678, (r2, A2), t>
6	<married_to, 1234, 5678>	<married_to, 1234, 5678>

Table 8.10. Triples and records for schema changes.

The *is-a* triples (lines 1 and 2) merely need altering to reflect that the entity has changed from *person* → *man* or *person* → *woman*. As the *is-a* triples are spread across the disk array, the changes and any re-ordering can be handled on each disk. The schema triple (line 3) is held in memory and is a simple change. The entity triple (line 6) requires no alteration. However, the biggest changes are required for the attribute records (lines 4 and 5). Each attribute record needs splitting so that the correct attribute(s) are saved with the respective entity identifier. This means deleting the original attribute and entering the two new ones. As with other data types, the last digit signifies which disk the attribute records are on and this will remain the same for all such cases. Note that such changes do not affect the string tables or lexical triples.

Generalisation—combining two or more entities into a single entity class—can also be achieved. In the example above, for instance, we might want to merge *man* and *woman* into *person*. The reverse of the actions described above would therefore be required. For more complex changes, it is more likely that a new schema would be described and the database re-created from the raw data [GUE92].

8.8 Other DBMS related issues

In this section we briefly mention other important areas that must be considered by the database designers—integrity, concurrency and security—and show how they can be adequately accommodated within our architecture; although these areas are not a main focus of this thesis.

8.8.1 Data integrity

These are aspects of database systems that have been well documented over the years. In our case, integrity constraints are considered as schema data (and held as schema triples) and are adequately catered for via the integrity constraints in list comprehensions discussed in [POU89]. An example follows where the existence of entity *person* and relation *name_of* is assumed. The integrity constraint is designed to ensure that there is no person with an unknown '?' or undefined '@' name. Integrity constraints are Boolean-valued functions with zero arity and must always evaluate to true—the empty list. They are used when data is being entered at population time or when later data is added to a working system.

```
must_have_name : → Bool;
must_have_name <= [ x ‖ x ← All_person &
                            (name_of x = @) or (name_of x = ?)] = [];
```

It is customary for standard systems to maintain transaction logs for day-to-day usage. If a system crash occurs inspection of the logs enables data to be recovered. In our system, copies of entity triples and string tables as part of our RAID mirroring scheme provide security, particularly in the case of corruption of a whole disk. There are also the additional parity RAID disks that can be used to re-constitute any of the other disks in the array to get the database back to a correct state.

8.8.2 Concurrent access

In today's large-scale multi-user systems, it is standard practice to use a combination of object locking and timestamping to maintain data consistency.
Our situation is no different. However, this only applies to the work of data entry personnel, it should not be allowed to affect browsing of data. The effects of data entry can be minimised by updates to sections of the files so that locking is applied at a record level rather than at a database level.

8.8.3 Security

Users require that their data be protected against unauthorised access and update. The 'value' of data varies between different systems: by value we mean the costs involved if data is disclosed or destroyed against the wishes of its owner. These considerations will have more bearing for some systems than others. In the domain of investigative systems, the personnel browsing the system will be trained officers and expected to seek connections between various items of data. Moreover, the phenomenon sometimes applicable to statistical databases—whereby unauthorised access to sensitive information is gained by counting records using set and Boolean operators and negation—does not apply in our case. The information is there to be manipulated using any number of hypotheses to arrive at various conclusions. However, protection must be provided against improper deletion of data—which can be built into integrity constraints.

8.9 Summary

In this chapter we have used two running example databases to show how database creation, population and maintenance are handled. The North Yorks crime database is built around real-life data used for training purposes by the North Yorks police force. It gives a good idea of the types of data used in our domain and how entities and attributes relate. We scale up the data in the crime database to arrive at a much larger and more realistic database.

Because we do not have access to the raw data from the crime database, we are unable to obtain an accurate estimate of the size of a larger database built using our architecture. Therefore we obtained the required information by extrapolation from the triples that constitute the crime database and create the records and triples that would exist if this data were stored using the new architecture. It is then possible to calculate the storage requirements and how they map to disks in the array. An important aspect of the development of our system is that object migration—generalisation, specialisation etc—must be seamlessly catered for. We show that using a combination of triples and records does not compromise object migration because the semantic freedom of object identifiers is maintained in our architecture. We also discuss an index scheme to add a temporal dimension, although there are many other ways to go about achieving this.

Chapter 9 Summary, conclusions and further work

9.1 Introduction

This chapter draws together all aspects of this thesis into a summary that contains the statement of the problem, alternative solutions considered, the solutions chosen with reasons, evidence to support the solutions and conclusions. Finally further areas of work in relation to this thesis are suggested.

9.2 Statement of the problem

The Triple Store Applications Research Project (TriStarp) was started in 1984 and led by Professor Peter King at Birkbeck College, University of London. The objective was to explore and develop the functional view of the binary relational approach as a database formalism and combine this with functional programming. The results were successful and the project has undergone several enhancements since then. In 1994 Professor Victor Maller joined Professor King in a collaborative project of which this thesis is a part. The storage sub-system underpinning the project since its inception has been a software triple store—discussed in detail in chapter 2. We highlight below areas where we believe there are important omissions from, or weaknesses in, the TriStarp proposals that this thesis aims to tackle.

An intrinsic belief from the outset of the TriStarp work has been the strict adherence to a triple store for all data. Clearly some data is best considered as triples and the concept nicely complements the functional model with its three-element data structure `<subject, relation, object>`. However, other data does not fit as easily into this structure. Strings are the main example and were discussed at length in chapter 4, but other data types—such as binary large objects—would be just as unwieldy broken down into a triple structure. We believe this distinction between the logical level of data and the physical level of data was an important area overlooked. Although the simplicity of a semantic-free interface is persuasive, we believe it is too rigid to permit the enhancements necessary to make the software a more attractive commercial prospect. We are not aware of any comparable system that uses triples for all data in the way described in chapter 2.

Another feature of the original proposals that was not given enough consideration was the functionality available for string handling. This is an area of data manipulation where functional languages have traditionally been somewhat weak. There are two distinct problems here. Firstly, the way strings are manipulated in functional languages—usually using list construction, head and tail etc—is often a slow and resource-consuming exercise. Secondly, the way strings are decomposed into triples as part of the homogeneous triple store makes them more difficult to work with. The constant re-construction of strings for comparison with search patterns, together with continually crossing the interface between levels 1 and 0, seriously impedes performance. Moreover, the significant number of tokens created for the comparisons, are otherwise useless and deplete the token space available for strings—sometimes to exhaustion—thus compromising data integrity.

Since the binary relational storage structure was first proposed by Frost as a means of underpinning Shipman's functional data model, there has not been agreement on the optimal way to store data. However, there seems to be a consensus that a balance needs to be struck between the degree of duplication desirable for rapid access of triples set against the increased costs of maintaining this duplication. Additionally, the way graph traversal operations are handled necessitates the sequential processing of each step for the 'start' (initial filter attributes) and 'stop' (display attributes) in a series of graph traversal steps.

Another significant topic for discussion is the applicability of a parallel implementation. This must be considered if the software is to handle large data sets and therefore be of realistic value with the potential of being used as the basis for a commercial product. The current architecture could be made parallel in a number of ways: these are assessed in chapter 3.

The more general topic of enhancing interface functionality is another area that requires investigation. The reduced set of semantic-free storage level interface functions provides a simpler interface for the model level language developer, but means there is less flexibility. This was in keeping with the triple store concept discussed above. However, functionality for handling data types—particularly strings—is a separate issue. There are possibilities to introduce functionality at a lower level in the query evaluation process.

Note there are other issues from the original proposals that may be alluded to but not discussed at length in this thesis. These include: providing richer data types, improving the user interface, improving the treatment of range queries, scoping of updates, handling unknown information and extracting schema information from partially structured data. These areas have been, or will be, the subject of other research.

9.3 The solution

Our solutions to the problems identified above and the alternatives considered are as follows.

A main theme of this thesis is that an alternative storage architecture is needed. One that combines the advantages of the triple store—best suited to storing binary relational data—with the benefits of holding data as collections of records, which underpins the still-popular relational data model. Chapter 3 considers alternatives to triple stores and their inherent indexing structures and concludes that a combination of the two alternative approaches is possible. We base our architecture on the ADMS data model. This makes it possible to store attribute data as record sets while leaving the entity-to-entity relationships and meta data as triples. Included as part of this scheme is the removal from records of attributes for some data types that are then stored separately. Text type and binary large object type are examples.

Coupled with the storage model for data is the use of a MIMD, dataflow model to permit parallel processing. MIMD was chosen because it has emerged as the *de facto* standard for distribution of instructions and data using a loosely coupled processor/memory configuration. Our scheme is based on that used by the Teradata Corporation—a proven leader in this field. Parallel MIMD machines are dominated by asynchronous events and these are more effectively handled by an interrupt-driven model rather than a model that uses polling. An interrupt is an example of data-driven scheduling which naturally complements the use of a dataflow model for computation and long latency operations.

Because of the referential transparency of functional languages, there are many ways in which they can be made parallel. We consider these before deciding upon dataflow, which naturally complements the functional approach and can extract orders of magnitude more parallelism from a functional language than from an imperative language. Moreover, dataflow used with a functional language obviates many of the control and communication overheads that exist in imperative languages.

The improvement in string manipulation is effected in two ways. Firstly, within the confines of the current architecture, we enhanced the functionality for string handling. This meant making available more functions to manipulate strings, which involved allowing search patterns to contain missing characters and/or left- and right-handed truncation. These functions do not compromise the benefits of using a functional approach as they are evaluated at the storage sub-system level and can thus be considered as built-in or object-level operators. However, this does not address the difficulty of how strings and search patterns are broken down into triples for storage and comparison.

Secondly, and in keeping with our strategy that not all data is best stored as triples, we discuss different data structures for strings. There are advantages and disadvantages in using a tokenised scheme. The merits of this are presented before describing our data structures for strings. By storing strings in a look up table more powerful functions can be provided to manipulate them. We describe some of these and generalise the argument for providing functionality in this way.

Improvements to general functionality are also possible. In chapter 7 we examine the work done with AGNA and extend it to support our string and text types in particular. With judicious optimisation of user queries and list comprehensions, it is possible to collect similar predicates into a generator/filter operation that can be passed down to the storage sub-system. This can be done safely so as not to compromise referential integrity or any other advantages of the functional paradigm.

As part of our architecture proposals, we discuss redundancy and show how this can be used effectively for our system. We introduce a novel RAID configuration that uses a combination of parity and mirroring. In each case the mirror disk is not a direct copy of the data disk but a combination of different data placement strategies for different data. For attribute records, meta data and lexical triples, it is an exact mirror. But for entity-to-entity (E→E) triples and string tables the data is duplicated in reverse order. This is explained in chapter 6. These alternative approaches to holding data complement the functional data model as well as functional languages. In both cases it is often necessary to traverse the graph model from range to domain—the inverse function. For E→E triples this can mean using the *inv_f* of the function *f*. For attribute data this can mean using a similar function or just using the function as a filter condition. For E→E triples, the alternative disks can be used to evaluate more quickly the application of a function. For string searches where the pattern has an unknown left-hand end and a known right-hand end, the inverse string tables can be used.

Finally, we compared using inverse function with conditions other than equality. At the moment equality is implied in inverse functions that map a constant to an entity identifier or set of entity identifiers. By making it necessary to include the operator (theta condition) as part of the expression, greater flexibility can be provided for inverse function operations.

9.4 Proof of solution

The choices made above and how they synthesise with our architecture are now discussed.

Combining records and triples

Our solution for holding a combination of triples and records for instance data is shown to work with no loss of information for the following reasons. Each attribute is ordered in a look up table for its type. Each attribute is safely mapped to the entity identifier. The record collections also hold the actual attribute values that can then be used for printing and display purposes. Bulk data, default information and missing information are an important part of the data model; chapter 6 shows how these are handled without compromising any information. Moreover, some data types—such as documents and binary large objects—are best kept in unfragmented format and are large enough to be held separately from the records with a link to maintain the connection.

The semantic freedom of entity identifiers is important to maintain. This is so object migration—i.e. generalisation and specialisation—can be accommodated seamlessly into the data (and storage) model. Chapter 8 shows this is still possible with our combined data model.

Because we do not have access to the raw data for the crime database, we had to extrapolate from the data as best we could. This was done to show how a standard database would be loaded, sorted and indexed for our architecture. In chapter 8 we use the crime database statistics to show how triples from this database would be held as records with all necessary identifier information added. As the crime database uses a rather small data set, we multiplied the extrapolated information by a factor of 200. This allowed us to demonstrate the triple/record allocation for a more realistic database.

String enhancements and data structures

The first stage involved adding string manipulating functionality within the confines of the triple store architecture. Chapter 4 sets out the experimental function that we developed. From these functions a more meaningful sub-set was produced for use in the crime pattern analysis work that is another area of research within the TriStarp project. The improvements of some orders of magnitude are not surprising as, using the new string functions, there is no need to cross the interface boundary so frequently. Moreover, there is no longer any need to create a mass of sub-strings—and concomitant tokens—that are only required for comparison with the search pattern. Because the functions are non-updating and can be regarded as object-level primitives, such as + and –, they can be safely included in user expressions and nested to the same depth without compromising integrity.

Using alternative data structures, where strings are no longer stored as triples, experimental programs were run against a larger data set of over 5 million records. Where the search pattern has a known beginning and/or ending, the search can take around 3 to 5 seconds with a search engine. Exhaustive searches of the strings, or where neither the beginning nor end of the search pattern is known, can take around 30 seconds with a search engine. These are basic figures for a uni-processor architecture only. As has been shown in this thesis, a parallel architecture reduces the search times accordingly.

Comparisons with SQL are difficult to make and not necessarily fair. This is because SQL uses far more comprehensive indexing techniques—often using complete names to provide very rapid entry into a database. The main advantage of our data structures is that the software can now handle far more powerful searching functions within the computationally more comprehensive functional paradigm.

Redundancy and inverse functions

Chapter 6 gives an example of how redundancy is used in our architecture. Using examples we show how triples are allocated to the data disk and the mirror disk. The search path used in the example has both normal and inverse function applications to show that no information is lost when executing a query. Bulk data and missing or unknown information are also catered for so the data model is not compromised. The data placement algorithm ensures there is an even distribution of triples across the array.

The combination of RAID technologies—mirroring and parity—substantially improves the mean time between failure (in theory to 34 billion years!). The throughput handled by a RAID 3 system can be dealt with in 60% of the time using our RAID 15 system. The price of average throughput for RAID 15 compares favourably with RAID levels 1 and 5; is significantly better than RAID 3 but is well below RAID 100. However using RAID 100—where data is written to the mirror disk in reverse placement order to that stored on the data disk—would not suit our architecture or data structures so well.

Inverse functions now require the test condition to be added to the expression. Chapter 7 gives an example of how this was used in the crime database and that an improvement of two orders of magnitude is possible in the case of equality. For other theta conditions, care must be taken to restrict the set of conditions available for each specific type used in the database. This has to be checked as part of the type system but, once done, these expressions can be passed directly to the storage sub-system for evaluation. This is discussed next.

MIMD and dataflow

Our use of a MIMD parallel machine configuration and dataflow model sensibly follows on from earlier research in these areas which has already proved successful commercially. Moreover, the concept of reducing functionality to a lower level in the query evaluation process was included in the AGNA project. We extend this to include additional functionality for string and text manipulation, inverse function applications and optimisations to complement our architecture. Our coarse-indexed access method does not severely compromise the advantages of non-indexed access, and at the same time does not add much to communications overheads. This is because of our use of set collation and open lists and is explained in chapter 6.

9.5 Conclusions

This thesis set out to investigate two specific areas of the continuing TriStarp project. String manipulation and graph traversal. Additionally, our investigation identified other areas of weakness in the functionality, plus constrictions imposed by the data model used. This led to the development of string manipulating functions based on well-known algorithms for inclusion in the current architecture. This proved highly successful but the addition of more powerful string manipulating functionality was still limited by the triple store architecture.

The next step was to show how strings could be taken out of the homogeneous triple store without compromising the functional paradigm. This was incorporated into a new architecture that combines the best of the functional data model with the best of the relational data model.

The architecture draws a distinction between the logical view of data as triples and the physical view of data as entity sets and attribute records. Again, this does not compromise the fundamental strengths of the data model. Included in this is a novel RAID configuration that complements the inverse functions and reverse graph traversal steps that are frequently used in functional programming and the functional data model respectively.

The final part of our strategy for a new architecture is the inclusion of parallel processing techniques to boost performance further. We adopt two tried and tested areas in this field—MIMD and dataflow—and show how our physical model and enhanced functionality can be easily accommodated by these techniques. The concept of devolving functionality is taken from earlier research work and enhanced. Timings are given that show the improvements achievable.

The conclusion of this thesis is that enhanced string manipulation and general functionality are possible without compromising the strengths of the data model and functional programming. Moreover, our storage model and use of redundancy combines the strengths of the functional and relational models with those of functional programming in a novel way. These are the areas of contribution in this thesis.

9.6 Further work

We are pleased with the findings from the areas of work investigated in this thesis. However, there are several areas that suggest further work.

9.6.1 Build complete system

An obvious step would be to build a complete system incorporating all the ideas introduced in this thesis. However, the best way to do this is not immediately clear. The current TriStarp model level languages are not written with dataflow or even parallelism in mind. This would mean a complete re-write of the compiler and parser would be needed, and this is a major undertaking. However, some concepts could be incorporated into the current software more easily than others.

The string manipulating functions, providing a much broader set of search options, were incorporated in the software as explained in chapter 4. A richer type system for text would allow further functionality to be added that could be targeted to the larger strings that are more loosely connected. For instance, a function that would "find string *a* and string *b* in cases where they are no more than *n* words apart and return the sentence(s) in which they appear" could be meaningful and practical for text objects.

Incorporation of function inverse facilities would be more complex to implement and would have implications for the parser. This is because the application of each specific operator—equals, not equals etc—needs careful checking and constraining so that predicates involving inverses for certain types only permit certain operators allowable for that type. For example, not equals when used with a string predicate might result in an unacceptably high hit-rate—although this would, of course, be conveyed to the user for confirmation to continue. But it might be decided beforehand to disallow this operation for strings.

9.6.2 Partially structured data

In chapter 4 we highlighted the growing need to differentiate between short strings that tend to form a close semantic unit and long strings that have looser semantics. As was made clear, this is a sensible thing to do and merely follows current database technology where free-text objects are increasingly used. However, in addition to a richer type system mentioned above, we believe more can be done in this area. In particular, concerning data that is partially formatted.

Partially formatted data is a combination of structured data—based around a pre-defined schema—and unstructured data held in its raw state. The term *partially structured data* seems a good description of such a combination and is used in the TriStarp work, although it should not be confused with the term semi-structured data which is taken to mean 'self-describing' data as found in [ABI00] for example. A police officer's scene of crime report, which may already be a field in the database, is an example of unstructured data—although to the police officer it might be considered as structured data. Such data is used in keyword searches and may be displayed or printed for human consumption but is not otherwise processed in the sense that nothing new is added to the database schema. We believe that it could be used to add semantics to the database and that this is an area for further investigation.

Applications that might benefit from this approach are those where alterations to both type and instance data are viewed as equally important. The database schema should be capable of evolving as increased semantics are drawn from the raw data although, in most cases, it is envisaged that the raw data remain unchanged by any schema alterations. For domains such as ours it is quite likely that a significant amount of initial information may be of a free-text form. By applying proven techniques from natural language processing and computational linguistics to the free-text, new types and instances would evolve and could be added to the database.

9.6.3 Optimisation, transformation and searches

In chapter 4 section 4.2.2 we gave comparisons between conjunctive and disjunctive searches in the context of using our multi-match functions and user-defined functions. The inference was that further work could be done in this area in relation to optimisations in functional languages.

Passing down expressions directly to the storage sub-system has already been discussed in relation to our work and the work of others. However, the passing down of different search terms embedded in the same search pattern has not, to our knowledge, been investigated before and could prove an interesting area for further research. There could be different ways of providing users with complex and powerful search strategies, perhaps in the form of regular expressions similar to those used in the UNIX operating system. Moreover, searching for several search terms in a text could be combined at a lower level in the evaluation process thus reducing search times further.

List comprehensions have, for several years, been the subject of various optimisation techniques. These frequently involve moving sub-expressions around in the overall expression to promote certain operations that—for example—reduce the search space more quickly. Our improved functionality and grouping of expressions with like terms follow this approach. However, when the database is used merely as a functional programming language and persistent data is not consulted, there seems to be little point in creating string tokens which are then added to the database string triples, when they are not ultimately part of the instance data.

A detection mechanism needs to be incorporated so that, if the database is used for non-updating computation, e.g. `337 + 989`, or simple string comparisons, e.g. `contains "trivia%" "this is a trivial sentence"`, these operations can be evaluated immediately. This would mean the database is not consulted and thus obviate the time-consuming *lexeme → token* mappings etc. A first step might be to scan the initial expression to see if any functions are used that form part of the database schema. If there are not, then an alternative evaluation should proceed.

Differentiating between expressions in this way has been achieved in the Prolog Functional Data Model language P/FDM [GRA92] which incorporates **function methods** and **action methods**. Function methods are used when there is no side effect: action methods are used where there are side effects.

9.6.4 Hybrid RAID systems

Our combination of parity and mirroring RAID system is novel in that the placement of triples is reversed on the mirror copy. However, there have been other combinations of parity and mirroring RAID systems developed in recent years [MAS97]. Some of these use virtual disks of either, striped and then mirrored or mirrored and then striped arrays. The consensus is that striping of mirrored arrays is the preferable option. The discussion in chapter 6 made it clear that we suggest using *both* mirroring and parity in the physical sense—as opposed to the virtual sense—to provide extra protection etc. However, what is not clear is which of the schemes to implement first—mirroring or parity. From the foregoing, it would appear that writing to the mirror array followed by the striping across (each) array is the better option. However, as our use of the mirror is unique for our architecture, further work is required to ascertain the best approach to adopt in our case.

References

[ABI00] S. Abiteboul, P. Buneman and D. Suciu.
Data on the web.
Morgan Kaufmann publishers, 2000.

[AKE93] G. Akerhold *et al.*
Processing transactions in a parallel functional language.
Proceedings of PARLE, Munich Germany 1993.

[ALB91] A. Albano *et al.*
A relationship mechanism for a strongly typed object-oriented database programming language.
Proc. 17th VLDB conference 1991, pp 565–575.

[ALP97] A. Alpkocak and E. Ozkarahan.
A spatial grid file for multimedia data representation.
4th International Conference on Parallel Computing Technologies, 1997 pp 156–167.

[ARG87] G. Argo *et al.*
Implementing functional databases.
Proc. Workshop on Database Languages, 1987, pp 87–103.

[ARV88] D. E. Arvind *et al.*
Assessing the benefits of fine-grained parallelism in dataflow programs.
Int. Journal of Supercomputer Applications, 2(3), 1988.

[AYR95] R. Ayres.
Enhancing the semantic power of functional database languages.
PhD Thesis, Birkbeck College, University of London, 1995.

[BAB79] E. Babb.
Implementing a relational database by means of specialized hardware.
ACM Transactions on Database Systems, vol. 4(1) 1979, pp 1–29.

[BAN88] F. Bancilhon.
Object-oriented database systems.
ACM symposium on principles of database systems, New York 1988, pp 152–162.

[BIE97] E.W. Biersack and C. Bernhardt.
A fault tolerant video server using combined RAID 5 and mirroring.
Proceedings of SPIE, vol. 3020 1997, pp 106–117.

[BOR90] H. Boral *et al.*
Prototyping Bubba, a highly parallel database system.
IEEE Transactions on Knowledge and Data Engineering, vol. 2(1) 1990, pp 4–24.

[BOY77] R. S. Boyer and J. S. Moore.
A fast string searching algorithm.
Communications of the ACM, vol. 20(10) 1977, pp 762–772.

[BUN82] P. Buneman *et al.*
An implementation technique for database query languages.
ACM Transactions on Database Systems 7(2) 1982, pp 164–87.

[CAF85] ICL Technical Journal, vol. 4(4) 1985.

[CAT94] R. G. G. Cattell.
The object database standard—ODMG-93.
Morgan Kaufmann publishers, 1994.

[CHA96] A. Chambers and J. Tidmus.
Practical parallel processing.
International Thomson, London, 1996

[CHU89] S. H. Chun *et al.*
A partitioning method for grid file directories.
Computer Software and Applications, 1989.

[COC98] W.P. Cockshott et al.
Data compression in database systems.
International Database Engineering and Applications Symposium, Cardiff 1998, pp 111–120.

[COD70] E. F. Codd.
A relational model for large shared data banks.
Communications of the ACM, vol. 13(6), 1970.

[COD79] E. F. Codd.
Extending the relational database model to capture more meaning.
ACM Transactions on Database Systems, vol. 4(4), 1979.

[CRA75] B. Cranston and R. Thomas.
A simplified recombination scheme for the Fibonacci buddy system.
Communications of the ACM, vol. 18(6), 1975.

[CRO92] L. E. Crockford and A. Drahota.
RIBA – A support environment for distributed processing.
ICL Technical Journal, vol. 8(2) 1992, pp 284–301.

[CRO82] L. E. Crockford.
Associative data management system.
ICL Technical Journal, vol. 3(1) 1982, pp 82–96.

[DER85] M. Derakhshan.
A hierarchical B+-tree approach to the implementation of a semantic free triple store.
Internal report of the TriStarp Group, MD/OCT/1985.

[DER89] M. Derakhshan.
A development of the grid file for the storage of binary relations.
PhD Thesis, Birkbeck College, University of London, 1989.

[DEW90] D. J. DeWitt and J Gray.
Parallel database systems: The future of database processing or a passing fad?
ACM SIGMOD Record, vol. 9(4) 1990, pp 104–112.

[DEW90a] D. J. DeWitt *et al.*
The gamma database machine project.
IEEE Transactions of Knowledge and Data Engineering, vol. 2(1) 1990, pp 44–62.

[DOT96] F. Dotsika and P. J. H. King.
The TriStarp common software manual.
Available from TriStarp Group, Birkbeck College, University of London, Malet Street, London, WC1E 7HX.

[EIS99] A. Eisenberg and J. Melton.
SQL:1999, formally known as SQL3.
ACM SIGMOD Record, vol. 28(1) 1999, pp 131–138.

[FEL69] J. A. Feldman and P. D. Rovner.
An ALGOL-bases associative language.
Communications of the ACM, vol. 1969, pp 439–449.

[FIE88] A. J. Field and P. G. Harrison.
Functional Programming.
Addison Wesley, 1988.

[FLY72] M. J. Flynn.
Some computer organisations and their effectiveness.
IEEE Transactions on Computers, vol. 21 (9) 1972, pp 948–960.

[FRE87] M. Freeston.
The BANG file: A new kind of grid file.
Proc. ACM SIGMOD conference, 1987.

[FRE89] M. Freeston.
Advances in the design of the BANG file.
Foundations of Data Organization and Algorithms.
Proc. 3rd International Conference, Paris, June 1989, pp 322–338.

[FRI96] M. Friedman.
RAID keeps going and going.
IEEE Spectrum, 1996.

[FRO82] R. A. Frost.
Binary relational storage structures.
The Computer Journal, vol. 25(3) 1982, pp 358–367.

[GAR92] H. Garcia-Molina and K. Salem.
Main memory database systems: An overview.
IEEE Transactions on Knowledge and Data Engineering, vol. 4(6) 1992, pp 509–516.

[GOL90] C. F. Goldfarb.
The SGML handbook.
Oxford: Clarendon Press, 1990.

[GRA92] P. D. Gray *et al.*
Object-oriented databases: A semantic data model approach.
Prentice Hall Series in Computer Science, 1992.

[GRU00] M. Gruber.
SQL Instant Reference (2nd Ed.).
Sybex, 2000.

[GUE92] R. Guest and P. J. H. King
Notes on specialisation, generalisation and inheritance in FDL.
TriStarp, Birkbeck College, University of London, 1991/92.

[GUT84] A. Guttman.
R-Trees: A dynamic index structure for spatial searching.
SIGMOD Record, vol. 14(2) 1984, pp 47–57.

[HEL78] G. D. Held and M. R. Stonebraker.
B-trees re-examined.
Communications of the ACM, vol. 21(2) 1978, pp 139–143.

[HEY91] M. L. Heytens and R. S. Nikhil.
List comprehensions in AGNA, a parallel persistent object system.
FPCA '91, Berlin, pp 569–591, 1991.

[HIL95] S. Hilditch.
RAID.
Ingenuity, vol. 10(1) 1995, pp 134–147.

[HIL1891] D. Hilbert.
Uber die stegie Abbildung einer Linie auf Flachenstuck.
Math. Ann. 38 1891, pp 459–460.

[HIN85] K. H. Hinrichs.
Implementation of the grid file: Design concepts and experience.
BIT, vol 25, 1985.

[HOR80] R. N. Horspool.
Practical fast searching in strings.
Software – Practice and experience, vol. 10, 1980, pp 501–506.

[HOS92] N. Hosur et al.
Dynamic addition and removal of attributes in BANG files.
Proc, ACM SIGAPP Symposium 1992, pp 210–216.

[IAN88] R. A. Iannucci.
A dataflow/Von Neumann hybrid architecture.
PhD Thesis, MIT, 1988.

[ILL96] R. Illman.
Re-engineering the hardware of CAFS.
ICL Systems Journal, vol. 11(1) 1996 pp 71–83.

[JUL97] J. Julliand and B. Markhoff.
Functional programming on MIMD multicomputers.
Int. Journal of Computers and Applications, vol. 19(3) 1997 pp 150–154.

[KAY85] M. H. Kay.
Textmaster – A document retrieval system using CAFS-ISP.
ICL Technical Journal, vol. 4(4) 1985, pp 455–467.

[KIM95] Sang-Wook Kim et al.
A new algorithm for processing joins using the multilvevel grid file.
Database Systems for Advanced Applications, 4th International Conference 1995, pp 115–123.

[KIM97] Sang-Wook Kim et al.
Linearity in directory growth of the multilevel grid file.
Information and Software Technology, vol. 39(13) 1997, pp 897–908.

[KIM98] Sang-Wook Kim et al.
Performance characteristics of the multilevel grid file.
Journal of KISS (Software and Applications), vol. 25(2) 1998, pp 239–252.

[KIN90] P. J. H. King *et al.*
TriStarp – an investigation into the implementation and exploitation of binary relational storage structures.
Proc. 8th BNCOD, York, 1990.

[KIN92] P. J. H. King and D. R. Sutton.
Fudal: A functional database language based on modal logic.
Actes du Congrès INFORSID '92, 1992.

[KIN96a] P. J. H. King and V. A. J. Maller.
Evaluating functional database concepts in advanced application environments.
Final report EPSRC research grants GR/K17736 and GR/K18313, 1996.

[KIN96b] P. J. H. King and R. Ayres.
Querying graph databases using a functional language extended with second order facilities.
Proc. 14th BNCOD, Edinburgh, 1996.

[KNU73] D. E. Knuth.
The art of computer programming: Vol. 3: Sorting and searching.
Reading Mass., Addison Wesley, 1973.

[KOC94] T. R. Kochtanek.
Standards for full text document storage.
Proc. 15th National Online Meeting, New York 1992, pp 301–307.

[LAV84] S. H. Lavington and C. Wang.
A lexical token converter for the IFS.
Internal report IFS/5/84, University of Manchester, 1984.

[LAV88] S. H. Lavington.
Technical overview of the Intelligent File Store.
Knowledge-Based Systems, vol 1, no 3, 1988.

[LAW00] J. Lawder.
An exploration of the application of space filling and other curves in multi-attribute indexing data.
PhD Thesis, Birkbeck College, University of London, 2000.

[LEE98] C. Lee and T-M. Tseng.
Temporal grid file: a file structure for interval data.
Data and Knowledge Engineering vol. 26(1998), pp 71–97.

[LEV67] R. E. Levien and M. E. Maron.
A computer system for inference execution and retrieval.
Communications of the ACM vol. 1967, pp 715–721.

[LIN96] R. D. Lins.
Functional programming and parallel processing.
Vector and Parallel Processing, 2nd International Conference, Porto 1996, pp 429–457.

[LOI97] H-W. Loidl and P. W. Trinder.
Engineering large functional parallel programs.
Implementation of Functional Languages, 9th Workshop 1997, pp 178–197.

[MAG80] D. R. McGregor and J. R. Malone.
The fact database: A system based on inferential methods.
Proc. of the Symposium on research and development in information retrieval, 1980, pp 203–217.

[MAG82] D. R. McGregor and J. R. Malone.
The fact database: A system using generic associative networks.
Information Technology: Research and Development, 1982, pp 55–72.

[MAL98] V. A. J. Maller and S. N. Sheldrake.
Storage sub-systems to support large functional databases.
International Database Engineering and Applications Symposium, Cardiff 1998, pp 52–53.

[MAL96] V. A. J. Maller.
Criminal investigation systems:
The growing dependence on advanced computer systems.
IEE Computing and Control Engineering Journal, vol. 7(2) 1996, pp 93–100.

[MAL88] V. A. J. Maller.
Outline description of the Teradata DBC/1012 database machine.
Used in lecture notes for database courses at Loughborough.

[MAL79] V. A. J. Maller.
The Content Addressable File Store.
ICL Technical Journal, vol. 1(3) 1979, pp 265–279.

[MAR84] N. J. Martin.
The construction of interfaces to triple based databases.
Proc. 3rd BNCOD, Leeds, 1984.

[MAR92] J. A. Mariani.
Oggetto: An object-oriented database based on a triple store.
The Computer Journal, vol. 35(2) 1992, pp 108–118.

[MAS97] P. Massiglia.
The RAID book: A storage system technology handbook.
Published by the RAID Advisory Board Inc., 1997.

[MCK92] R. L. McKee and J. Rodgers.
N-ary verses binary data modelling: A matter of perspective.
Data Resource Management, vol. 3(4) 1992, pp 22–32.

[MCN90] D. J. McNally *et al.*
Persistent functional programming.
Proc, 4th International Workshop on Persistent Object Systems, pp 59–90, 1990.

[MEA92] C. T. Meadow.
Text information retrieval systems.
Academic Press Inc., 1992.

[MEL02] J. Melton and A. R. Simon.
SQL:1999 – Understanding Relational Language Components.
Morgan Kaufmann publishers 2002.

[MER98] P. F. Meredith and P. J. H. King.
Scoped referential transparency in a functional database language with updates.
Proc. 16th BNCOD, Cardiff, 1998.

[MIT76] R. W. Mitchell.
Content Addressable Filestore.
Pro. Online Database Technology Conference, London, 1976.

[MOR99] J. Morris.
Parallel efficiency: The dataflow advantage.
4th Int. Symposium on Parallel Architectures 1999, pp 356–361.

[MUN78] P. Munz.
The WELL system: A multi-user database system based on binary relationships and graph pattern matching.
Information Systems, vol. 3 1978, pp 99–115.

[NAJ99] W. A. Najjar et al.
Advances in the dataflow computational model.
Parallel Computing, vol. 25 1999, pp 1907–1929.

[NIE84] J. Nievergelt, H. Hinterberger and K. C. Sevcik.
The grid file: An adaptive, symmetric multikey file structure.
ACM Transactions on Database Systems, vol. 9(1) 1984, pp 38–71.

[OUK85] M. Ouksel.
The interpolation based grid file.
Proc. 3rd ACM SIGACT-SIGMOD Symposium on principles of database systems, 1985.

[OUK92] M. Ouksel et al.
Concurrency control in the interpolation-based grid file.
Proc. DEXA conference 1992, pp 237–243.

[OUK94] M. Ouksel et al.
Management of concurrency in interpolation-based grid file organization and its performance.
Information Sciences, vol. 78 1994, pp 129–158.

[OZA96] K. Ozaki and Y. Yano.
The 3-tuple data modeling.
Proceeding of the IEEE International Conference on Systems, Management and Cybernetics, vol. 3 1996, pp 2149–2154.

[OZK85] E. A. Ozkarahan and M. Ouksel.
Dynamic and order preserving data partitioning for database machines.
Proc, 11th VLDB conference, Stockholm, 1985.

[PAG92] J. Page.
A study of a parallel database machine and its performance: The NCR/Teradata DBC/1012.
Proc. 10th BNCOD, Aberdeen 1992, pp 115–137.

[PAP95] A. Papantonakis and P. J. H. King.
Syntax and semantics of Gql, a graphical query language.
Journal of Visual Languages and Computing vol. 6, 1995, pp 3–25.

[PAT88] D. Patterson *et al.*
A case for redundant arrays of inexpensive disks (RAID).
Proceedings of 1988 ACM SIGMOD, June 1988.

[PEY87a] S. L. Peyton Jones *et al.*
GRIP – A high performance architecture for parallel graph reduction.
In 'FPCA' 87, 1987, pp 98–112.

[PEY87b] S. L. Peyton Jones.
The implementation of functional programming languages.
Prentice Hall International, 1987.

[PEY89] S. L. Peyton Jones.
Parallel implementations of functional programming languages.
The Computer Journal, vol. 32(2) 1989, pp 175–186.

[PEY93] S. L. Peyton Jones and P. Wadler.
Imperative functional programming.
ACM Symposium on Principles of Programming Languages, Charleston, 1993, pp 71–84.

[PEY96] S. L. Peyton Jones *et al.*
Concurrent Haskell.
Proc. 23rd ACM Symposium on Principles of Programming Languages, Florida, 1996.

[POU89] A. Poulovassilis.
The design and implementation of FDL, a functional database language.
PhD Thesis, Birkbeck College, University of London, 1989.

[POU92] A. Poulovassilis.
The implementation of FDL, a functional database language.
The Computer Journal, vol. 35(2) 1992, pp 119–128.

[PRO98] B. J. Procter.
The enterprise datacentre—ICL's 'millennium' programme.
ICL Systems Journal, vol. 13(1) 1998, pp 17–34.

[QUA99] F. Quaglia and B. Ciciani.
Performance vs. cost of redundant arrays of inexpensive disks.
Simulation Practice and Theory, vol. 7(2) 1999, pp 153–170.

[ROB89] I. B. Robertson.
Hope$^+$ on Flagship.
Proc. of the Glasgow workshop on functional programming, Glasgow 1989, pp 296–307.

[SAG94] H. Sagan.
Space-filling curves.
Sprinter-Verlag, 1994.

[SHA88] G. C. H. Sharman and N. Winterbottom.
The universal triple machine:
A reduced instruction set repository manager.
Proc. 6th BNCOD, Cardiff, 1988.

[SHA78] G. C. H. Sharman and N. Winterbottom.
The data directory facilities of NDB.
Proc. 4th VLDB Conference, Berlin, 1978.

[SHI81] D. W. Shipman.
The functional data model and the data language DAPLEX.
ACM Transactions on Database Systems, vol 6, no 1, 1981.

[SIL01] L. Silverston.
The data model resource book, vols. 1 and 2.
Wiley Computer Publishing, 2001.

[SMA88] C. Small.
Guarded default databases: A prototype implementation.
Prolog and databases: Implementations and new directions, 1988.

[SMI87] P. D. Smith and G. M. Barnes.
Files and databases: An introduction.
Addison Wesley, 1987.

[SOU97] S. B. Southerden.
INDEPOL Client—A 'facelift' for mature software.
ICL Systems Journal, vol. 12(2) 1997, pp 315–329.

[STO86] M. Stonebraker.
The case for shared nothing.
IEEE Data Engineering, vol. 9(1) 1986, pp 4–9.

[SU88] S. Y. W. Su.
Database computers, principles, architectures and techniques.
McGraw-Hill International, 1988.

[TAG85] R. M. Tagg.
CAFS-ISP: issues for the application designer.
ICL Technical Journal, vol. 4(4) 1985, pp 402–418.

[TIT74] P. Titman.
An experimental database system using binary relations.
Proc. IFIP TC-2 working conference, Amsterdam, 1974.

[TRI89] P. W. Trinder and P. L. Wadler.
Improving list comprehension database queries.
Proc. TENCON, 4th IEEE Conference, Bombay, pp 186–192, 1989.

[TRI96] P. W. Trinder *et al.*
GUM: A portable parallel implementation of Haskell.
Proc. Programming Language Design and Implementation, Philadelphia, pp 79–88, 1996.

[TRI98] P. W. Trinder *et al.*
Algorithm + strategy = parallelism.
Journal of Functional Programming vol. 8(1), 1998, pp 23–60.

[TRI00] P. W. Trinder *et al.*
The multi-architecture performance of the parallel functional language GpH Glasgow Parallel Haskell.
Euro-Par 2000 6th International Conference, 2000, pp 739–743.

[WHA85] K. Whang and R. Krishnamurthy.
Multilever grid files.
IBM research report RC11516 (#51719), 1985.

[WHA91] K. Whang and R. Krishnamurthy.
The multilevel grid file –
A dynamic hierarchical multidimensional file structure.
Database Systems for Advanced Applications, 1991.

[WIL00] S. Williams.
The associative model of data.
Published by Lazy Software Limited, 2000.

[WIL96] R. Wilhelm *et al.*
Parallel implementation of functional languages.
Analysis and Verification of Multiple-agent Languages, Stockholm 1996, pp 279–295.

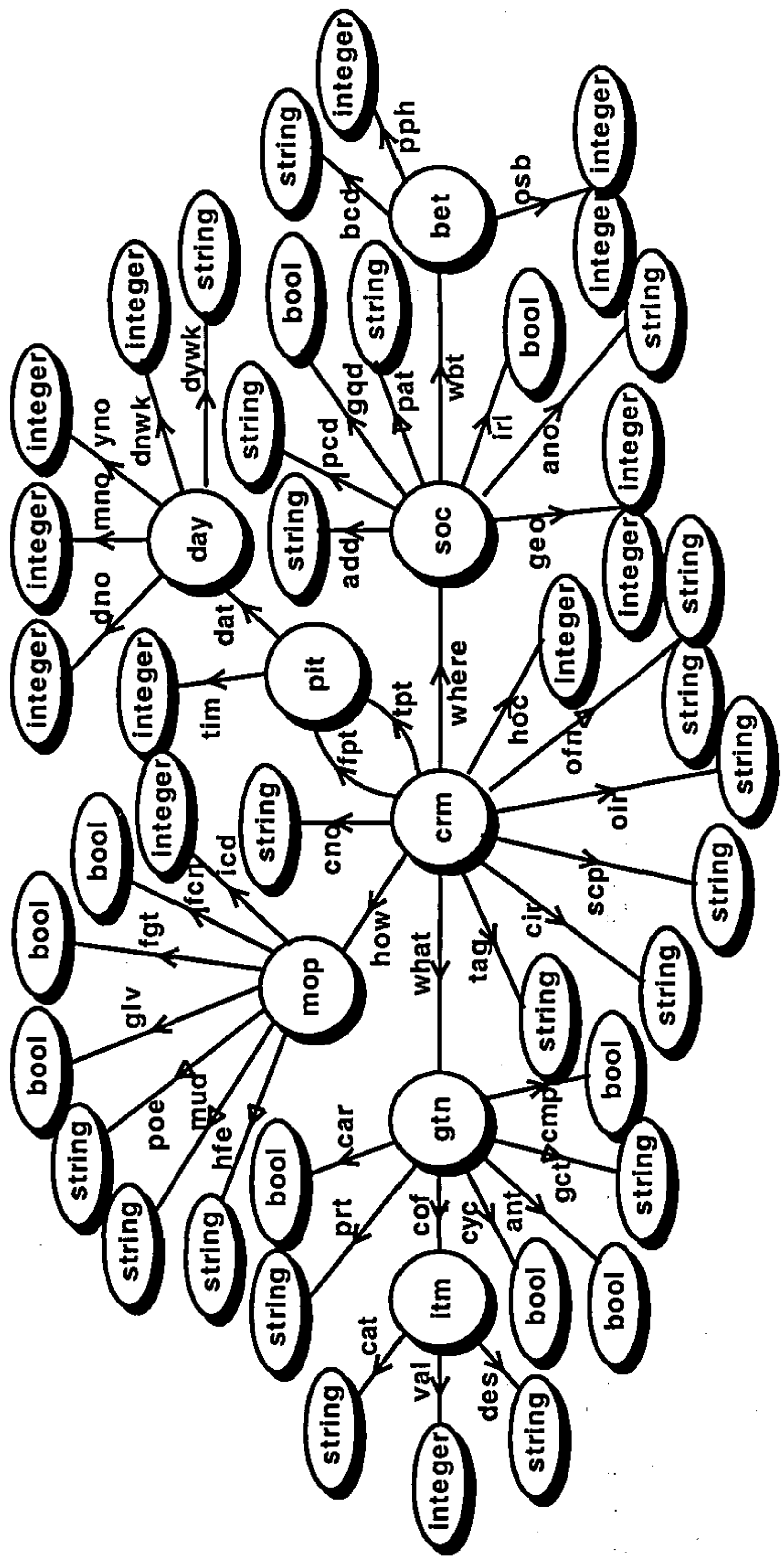
[WIL85] P.R. Wiles.
Using secondary indexes for large CAFS databases.
ICL Technical Journal, vol. 4(4) 1985, pp 419–440.

[WIN79] N. Winterbottom and G. C. H. Sharman.
NDB: Non-programmer data base facility.
IBM UK Laboratories Ltd, Technical Report TR.12.179, 1979.

[WIT93] A. Witkowski et al.
NCR 3700 – The next-generation industrial database computer.
Proc. 19th VLDB Conference, Dublin, 1993 pp 230–243.

Appendix

A.1 North Yorks crime database – schema



A.2 North Yorks crime database – triple allocation

After creating the database and adding all schema details	6,170

Membership triples

crm	2,501	itm	8,598	pit	5,002	
soc	2,501	gtn	2,501	mop	2,501	
day	2,770	bet	147			26,521

Entity-Entity Triples

gtn	→	itm	8,598	crm	→	gtn	2,501	
crm	→	mop	2,501	crm	→	soc	2,501	
crm	→	pit	5,002	soc	→	bet	2,501	
pit	→	day	5,002					28,606

Directory Pages – 26 pages @ 382 triples per page	9,168

String Triples

crm	→	tag	2,501	itm	→	cat	8,700	
	→	cir	69,000		→	des	118,000	
		+	2,501			+	8,598	
	→	scp	314,000	mop	→	hfe	2,700	
		+	2,501		→	mud	4,400	
	→	cno	5,002		→	poe	9,800	
	→	oir	2,501	soc	→	pat	6,800	
	→	ofn	4,400		→	pcd	2,501	
gtn	→	prt	2,500		→	add	14,000	
	→	gct	6,700			+	2,501	
bet	→	bcd	147	day	→	dywk	2,700	
							around	592,000

Non-String Triples

crm	→	hoc	2,501	itm	→	val	8,598	
gtn	→	ant	2,501	mop	→	glv	2,501	
	→	car	2,501		→	fgt	2,501	
	→	cyc	2,501		→	fcn	2,501	
	→	cmp	2,501		→	icd	2,501	
day	→	dno	2,770	bet	→	pph	147	
	→	mno	2,770		→	osb	294	
	→	yno	2,770	soc	→	irl	2,501	
	→	dnwy	2,770		→	gqd	2,501	
								47,630

Crime Pattern Analysis Triples

pat_bools	2,502 x 18	mud_bools	2,502 x 22	
hfe_bools	2,502 x 16	poe_bools	2,502 x 24	
gtn_bools	2,502 x 24			260,208

A.3 Presented at IDEAS '98 Conference, Cardiff UK, July 1998

Storage Sub-systems to Support Large Functional Databases

V.A.J. Maller and S.N. Sheldrake
Department of Computer Studies, Loughborough University, Loughborough, UK
email: [v.a.j.maller|s.n.sheldrake]@lboro.ac.uk

Abstract

Earlier work by V.A.J. Maller, and P.J.H. King eva'luated the suitability of a functional database language (FDL) being used to support large applications in the field of investigative systems [1]. This is a growing generic application area covering criminal and military intelligence and characterised by: significant data complexity; large data sets; and the need for high performance, interactive use [2].

The evaluation confirmed the soundness of FDL but, heavy use in a practical context, showed improvements were needed to areas like string matching and graph traversal. Also, an implementation on multiprocessor, parallel architectures would help meet the performance requirements arising from existing and projected database sizes in this application area.

This paper discusses some of the proposed changes to the interfaces in the software architecture for a future system that should meet both the requirements of extended functionality and potential parallelisation.

1. Background

The functional database language (FDL) [3] was developed as part of the TriStarp (Triple Store application research project) at Birkbeck College, University of London. The objective was to explore and develop the binary relational approach as a common framework – from the storage level, through the data model level, to the user level. A uniform set of functions provides the interfaces between the three levels.

Frost [4] reviewed the binary relational approach and devised the Binary Relational Storage Structure (BRSS). A BRSS holds data in three fields with the format: `<subject,relation,object>` termed a triple. A triple can hold simple facts like: "Fred reads The-Times", or be used for structures like binary trees. In this case, a set of triples with the format: `<node,left-subtree,right-subtree>` is used.

There were several developments of the BRSS concept and the Birkbeck Triple Machine (BTM) is one such implementation [5]. The BTM underpins the storage level of the TriStarp work and provides persistence for all data.

2. The current position

The BTM comprises two components: a software triple store and a lexical token converter (LTC). The triple store provides the storage mechanism via the following sets of interface functions: file utility operators – *create_db, open_db* and *close_db*; update operators – *insert_triple* and *delete_triple*; and retrieval operators – *open_set, close_set, fetch_another* and *present.*

The LTC handles the mapping between elements of a triple – strings, integers, etc. termed lexemes – and their unique, fixed-length (32-bit) token identifiers used to represent them in the triple store. Entities – termed non-lexicals – are represented by unique surrogates and are not directly visible to the user.

Using tokens saves duplicating strings (which form the bulk of the lexemes) and ensures compact storage in memory. However, most strings need decomposing into a set of triples because one triple can accommodate only six characters. Therefore reconstruction of strings into their full representation adds significantly to response time – particularly when handling large data sets.

Metadata and manipulation functions, like *map,* are stored in binary tree triples and clustered around unique identifiers where possible. However, triple retrieval and function reconstruction can add further overheads to response time.

As no semantics are attached to triples, the BTM is used effectively by both functional and logical databases. The decision to use a semantic-free triple store was in keeping with other research at the time; it was a deliberate attempt to keep the storage mechanism simple by providing a small set of interface functions. This means the majority of core database tasks like *select* and *join* have to be programmed at level 1 which can make them slow and difficult to optimise.

3. Research activities

The following are being undertaken:

- enhancing the interface functions,
- restructuring the physical model for data storage,
- incorporating additional hardware, and
- structuring changes to allow for parallel processing.

To improve string matching and graph traversal operations, the homogeneous triple store needs reworking so that data storage complements data usage. This will aid devolved functionality – for example, metadata would be bulk loaded into memory at the start of each session. The language Hydra [6], developed to evaluate associational features, highlighted two types of function definition: primary functions that hold instance data, and secondary functions – like *map* or *fold* – that manipulate instance data. These can be stored using different physical models.

Primary functions can be further subdivided into extensional and intensional varieties. The former are for simple `<subject,relation,object>` triples and are many in number; the latter are for equations involving expressions or variables on either side of their definition (such as default values) and are few in number, e.g.:

age Bill <= 47	extensional definition
age Joe <= age Eve	intensional definition
age x <= 30	default intensional definition

Surrogate tokens will be used for entity instances and relation names only: strings will be held in full. Although this results in some duplicated data, there are benefits where string matching is concerned.

The update operators *insert_triple* and *delete_triple* need semantics so the store manager will know where to store or locate them. We propose a richer set of interface functions for update operations, e.g.:

insert_pfun_extdef	extensional primary function
insert_pfun_intdef	intensional primary function
insert_sfun_def	secondary function

The retrieval operators would be enhanced by providing different levels of retrieval and associational functions. We plan to store triples with format: `<entity,relation,attribute>` in *cliques* grouped around the common entity. Parallel processing and data filtering techniques can then be used to boost performance levels. The entity triples with format: `<entity,relation,entity>` will remain, as these are used for graph traversal operations. So the retrieval choices proposed are:

get_triple	for one <E, r, A> triple
get_attributes	for <E, r, {A}> triples
get_Etriples	for <E, r, {E}> triples
get_all_from	for <E, r, {E}> + <E, r, {A}>
get_all_to	the inverse of the *get_all_from*
get_paths	retrieves paths from E_n to E_m

Restructuring the physical data model is now underway.

4. Conclusions

We believe these improvements will: provide a more coherent software architecture; improve performance levels and; support devolved functionality via a parallel processing array or a mature search engine such as ICL's CAFS [7]. After completion of this study phase, an operational system will be built which, together with an enhanced language FDL2, will be evaluated in an operational environment.

Acknowledgements

[1] had the support of EPSRC, ICL and the North Yorkshire Police Force.

Bibliography

[1] P.J.H. King and V.A.J. Maller, "Evaluating functional database concepts in advanced application environments", *Final Report EPSRC Research Grants GR/K17736 and GR/K18313*, 1996.

[2] V.A.J. Maller, "Criminal investigation systems: The growing dependence on advanced computer systems", *IEE Computing and Control Engineering Journal*, vol. 7(2) 1996, pp 93-100.

[3] A. Poulovassilis, "The implementation of FDL, a functional database language", *The Computer Journal*, vol. 35(2) 1992, pp 119-128.

[4] R.A. Frost, "Binary relational storage structures", *The Computer Journal*, vol. 25(3) 1982, pp 358-367.

[5] M. Derakhshan, "A development of the grid file for the storage of binary relations", PhD Thesis, Birkbeck College, 1989.

[6] R. Ayres and P.J.H. King, "Querying graph databases using a functional language extended with second order facilities", *14th BNCOD*, Edinburgh 1996, pp 189-203.

[7] V.A.J. Maller, "The Content addressable file store", *ICL Technical Journal*, vol. 1(3) 1979, pp 265-279.

A.4 Presented at BNCOD, Exeter UK, July 2000

Using Functional Databases to Support Large, Text-Intensive Applications

Simon N. Sheldrake and Victor A. J. Maller

Department of Computer Science,
Loughborough University,
Loughborough, LE11 3TU, UK
S.N.Sheldrake@lboro.ac.uk
V.A.J.Maller@lboro.ac.uk

Abstract. Functional languages, despite their many advantages, are not always seen as a first choice for database applications involving extensive string matching. One reason for this is their relatively poor performance in this area. This paper shows a way to resolve this without compromising the advantages of the functional paradigm.

Background

Earlier research evaluated a functional database, built over the functional data model, in the domain of investigative systems. This is a growing generic application domain covering areas such as criminal and military intelligence, which is characterised by significant data complexity, large text-intensive data sets and the need for high performance interactive use (Mal96). The evaluation confirmed the soundness of the functional approach but heavy use in a practical context showed weaknesses in crucial areas, particularly in string manipulation and graph traversal. This paper tackles the first of these.

The functional database language FDL (Pou92) was developed as part of the TriStarp project (Kin90) which set out to explore and develop the binary relational approach as a common framework for database design. FDL has as its model the functional view of the binary relational model, usually known as the functional data model or entity-function model. Such models view the world as consisting of non-lexical (abstract) entities—person, crime, etc.—and lexical (scalar) attributes—integer, string, etc.—used to describe the entities. Advantages of entity-function models include:

- intuitive and incremental schema evolution
- the ability to underpin other models—including the object-oriented (Ban90)
- ease of use provided by a graphical query language
- graphical representation of objects which may be viewed at various levels of abstraction.

The benefits of functional programming include:

- the ability to define complex data structures and recursive functions over them
- freedom from side effects because of referential transparency
- freedom of execution order facilitating parallel processing.

FDL extended these to include:

- persistence for all data
- the ability to define extensional and intensional definitions over the same function
- improved handling of null and undefined values—thus enabling database closure.

Persistence for all data is provided via a software triple store where all data is mapped to 32-bit tokens. This aids query evaluation and for most cases is very efficient. However, with strings, there are overheads imposed that may be tolerable in small-scale databases but are unacceptable for practical applications. The particular problems in our context are:

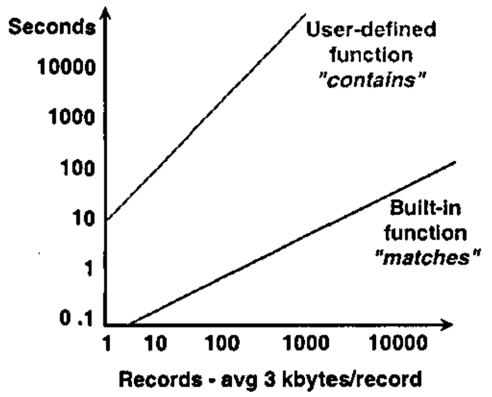
- recursion makes string functions slow to execute
- comparing strings has to be done using their tokens
- superfluous tokens are created during searches—sometimes to the point of exhaustion which compromises data integrity
- allowing for missing or unknown characters in the search pattern is non-trivial.

The aim is to improve string manipulation without losing the aforementioned benefits of a functional language—simply using another language would clearly compromise these benefits. The next section shows how improved string manipulation within the current architecture was achieved. Then, with different data structures for strings, it is shown how more can be done to provide users with the kind of facilities found in systems more traditionally associated with string handling.

Improvements to the Current System

String manipulation was accomplished by declaring functions in the language itself—at the user-level. Such functions, however, only had three built-in functions available for use in their algorithms—*concat*, *length* and *substr*. As user-level functions are recursive, each function call resulted in additions to the query evaluation tree and the creation of more tokens at each stage of the search. Built-in functions, on the other hand, merely 'drop out' of the evaluation tree, execute and return a single result. To resolve this, new built-in functions have been added that are fully compatible with a functional language. These functions search for space-delimited words and permit missing characters, left-hand or right-hand truncation of a search pattern and multiple search patterns. The new built-in function, *matches*, was compared to the FDL user-defined function, *contains*, and run against a trial database holding test data derived from real-world events. The results are shown below.

Triple store improvements



Alternative Data Structures

Although the above represents a substantial improvement, it is still difficult to provide users with comprehensive searching facilities. It is proposed, therefore, to take strings out of the homogeneous triple store so they can be more easily manipulated. Holding entity attributes (including strings) in their full format and clustered on their common entity, provides a physical storage structure combining the advantages of the relational model with those of the entity-function model. With strings delimited to word level and held in a look-up table, the important inverse mapping from attribute to entity can be made. Examples of these structures are: (where # indicates a uniquely generated word token.)

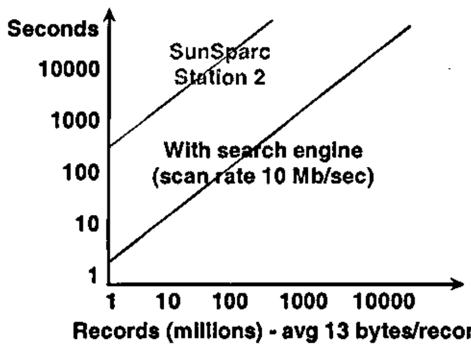
string	length	token	occurrences
analyst	7	#49	2,195
programmer	10	#3741	74,545

The occurrences field allows the user to ascertain quickly whether or not they wish to continue with the current search. With all attributes clustered on their common entity, the token-to-string mapping is unnecessary. Moreover, these structures are more amenable to search acceleration using a search engine such as CAFS (Mal79). From the above table the string triples might be: (where $ indicates an entity surrogate.)

token	relation	entity
#49	has_skills_as	$343218
#49	job_title	$923432
#3741	has_skills_as	$123456
#3741	has_skills_as	$128332

The ability to include missing characters was built into the search algorithm with the wildcards "_" – match any one character and, "%" – match zero or more characters. An example program demonstrated the speed-up that is possible: the results are shown below.

Improvements with clustered structures



References

Ban90 Bancilhon, F. *et al.*
The O_2 query language syntax and semantics.
Technical Report, Altair Paris, 1990.

Kin90 King, P. J. H. *et al.*
TriStarp – An investigation into the implementation and exploitation of binary relational storage structures.
Proc. 8th BNCOD, 1990.

Mal96 Maller, V. A. J.
Criminal investigation systems: The growing dependence on advanced computer systems.
Computing and Control Engineering Journal, vol. 7(2) 1996.

Mal79 Maller, V. A. J.
The content addressable filestore.
ICL Technical Journal, vol. 1(3) 1979.

Pou92 Poulovassilis, A.
The implementation of FDL, a functional database language.
The Computer Journal, 35(2) 1992.

A.5 'C' program files for text searching based on using records

```
/* File name : headers.h */
/* Files which use this are: main.c pat.c load.c search.c  */

#define  MAXSTRING        30       /* max length of any string */
#define  MAXLENGTH        2000     /* max number of entries in table */
#define  BIGLOOP          0        /* used for stats purposes */
#define  LOOPMAX          4000     /* used for stats purposes */

typedef  unsigned char    uchar;
typedef  unsigned short   ushort;

/* function prototypes  */

int     get_pattern  (uchar *pattern, ushort *patlen);
int     load         (uchar *index_char, char *file_name);
void    reverse_pat  (uchar *s);
void    fold_index   (void);
void    fixed        (uchar *pat, ushort plen);
void    front        (uchar *pat, ushort plen, ushort search_type);
void    mid          (uchar *pat, ushort plen, short wild);
void    midlast      (uchar *pat, ushort plen, short wild,
                      ushort search_type);
void    bothends     (uchar *pat, ushort plen);
void    midmid       (uchar *pat, ushort plen, short w1, short w2);
int     exact        (uchar *p, uchar *t, short i);
int     elastic      (uchar *p, uchar *t, ushort patlen);

/***********************************************************************/
/* enumeration used to classify search patterns, thus:  | with:       */
/*                                                      |             */
/* NONE = "........."  L = "........%"  M = "....%...." | F = first   */
/*   ML = "....%...%"  F = "%........" FL = "%.......%" | L = last    */
/*   FM = "%...%...." MM = "..%...%.."                  | M = mid     */
/***********************************************************************/

enum pattern_type {NONE, L, M, ML, F, FL, FM, MM};

struct record {              /* used for each entry in the table */
  ushort     length;             /* string length */
  uchar      *string;            /* actual string */
  unsigned   token;              /* id for the string */
  ushort     occurrences;        /* how many triples it appears in */
} entry[MAXLENGTH];

short wild[3];      /* place holders for any '%' wildcards in pattern */
```

```
/* File name : main.c */

#include "stdio.h"
#include "time.h"
#include "headers.h"

int alpha_index[27] = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
                        -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
MAXLENGTH};

unsigned long hits;

void main (void)
{
  uchar       pattern[MAXSTRING], *p;
  ushort      patlen, in_index;
  enum pattern_type  search_pat;

#if BIGLOOP
  time_t start, end;  /* these are used for */
  unsigned i;         /* stats purposes     */
#endif

  search_pat = get_pattern(pattern, &patlen);

  switch (search_pat) {
    case 0: case 1: case 2: case 3: case 7:
      in_index = load(&pattern[0],"forward.txt");
      if (!in_index) {
        printf("Not in table\n");
        exit(1);
      }
      fold_index();
      break;
    case 4: case 6:
      in_index = load(&pattern[patlen-1],"reverse.txt");
      if (!in_index) {
        printf("Not in table\n");
        exit(1);
      }
      reverse_pat(pattern);
      fold_index();
      break;
    case 5:
      load(NULL,"forward.txt");
      break;
  }

  p = pattern;
  while(*p) *p++ = toupper(*p);

  hits = 0;

#if BIGLOOP
  start = time(NULL);
  for(i=0; i<LOOPMAX; i++) {   /* do search x times for effect */
#endif
```

```
  switch (search_pat) {
    case 0: fixed(pattern, patlen);
            break;
    case 1: front(pattern, patlen-1, 1);
            break;
    case 2: mid(pattern, patlen-1, wild[0]);
            break;
    case 3: midlast(pattern, patlen-2, wild[0], 3);
            break;
    case 4: front(pattern, patlen-1, 4);
            break;
    case 5: bothends(pattern, patlen-2);
            break;
    case 6: midlast(pattern, patlen-2, patlen - 1 - wild[1], 6);
            break;
    case 7: midmid(pattern, patlen-2, wild[0], wild[1]);
            break;
  }

#if BIGLOOP
  }
  end = time(NULL);
  printf(" The table took %d second(s) to search\n",
                                            (int) difftime(end, start));
#endif

  if (hits) printf(" %d matches found\n", hits);
  else      printf(" Pattern not in table\n");
}

void reverse_pat
  (uchar *s)
{
  uchar c, i = 0, j = strlen(s) - 1;

  for (; i<j; i++, j--) {
    c = s[i];
    s[i] = s[j];
    s[j] = c;
  }
}

void fold_index
  (void)          /* if any index entries are still -1 they */
{                 /* need changing to reflect the start of   */
  short i = 25;         /* the next letter.                    */

  for (; i >= 0; i--)
    if (alpha_index[i] == -1) alpha_index[i] = alpha_index[i+1];
}
```

Appendix

```
/* File name : pat.c */

#include "stdio.h"
#include "headers.h"

int get_pattern
  (uchar  *pattern, ushort *patlen)
                               /* Gets pattern > 3 characters long with  */
{                              /* 0, 1 or 2 wildcards in it.  Returns to */
  uchar  i, total;             /* "patlen", and the enumerated integer   */
  uchar  ok = 0;               /* for the search strategy to be used.    */
  uchar  c, *w;

  do {                 /* continue until we have a pattern which is ok */
    w = pattern;
    printf("enter pattern length 4 or greater : ");
    while (isspace(c=getchar()))
      ;                  /* skip leading while space */
    *w++ = c;
    while ((*w = getchar()) != '\n')  /* now get pattern until CR hit */
      w++;
    *w = NULL;           /* append null byte */
    *patlen = strlen(pattern);   /* and get length */
    if (*patlen < 4) printf("too small - re-enter\n");
    else {
      wild[0] = wild[1] = wild[2] = -1;/* holds positions of wildcards*/
      for (i=0, total=0; total<3 && pattern[i]; i++)
        if (pattern[i] == '%') wild[total++] = i;
      switch (total) {  /* total now = number of wildcards in pattern */
        case 0: return NONE;
        case 1: ok = 1;
                break;
        case 2: if (wild[1] - wild[0] == 1)
                                  /* adjacent %s + pattern short*/
                  if (*patlen == 4)
                    printf("short pattern - re-enter\n");
                  else {              /* remove one of the wildcards */
                    for (i=wild[1]; pattern[i]; i++)
                      pattern[i] = pattern[i+1];
                    (*patlen)--;
                    wild[1] = -1;
                    ok = 1;
                  }
                else ok = 1;
                break;
        case 3: printf("too many wildcards - try again\n");
                break;
      }
    }
  } while (!ok); /* pattern has zero, one or two wildcards only in it */

  if (wild[1] == -1) {         /* ie, only one wildcard in pattern */
    if (wild[0] == *patlen-1) return L;
    if (wild[0] > 0) return M;
    else return F;
  }
  else                                /* two wildcards in pattern */
    if (wild[1] == *patlen-1)
```

```
      if (wild[0] > 0) return ML;
      else return FL;
    else if (wild[0] == 0) return FM;
         else return MM;
}

########################################################################

/* File name : load.c */

#include "stdio.h"
#include "headers.h"
#include "stdlib.h"

extern int alpha_index[];
unsigned count = 0;

int load
  (uchar *index_char, char *file_name)
{
  FILE *fp;
  uchar str[100], temp[20], last_char = '@', this_char;
  unsigned j, k;

  if ((fp = fopen(file_name,"r"))==NULL) {
    printf("cannot open in file\n");
    exit(1);
  }

  count = 0;

  while (!feof(fp)) {
    fgets(str, 100, fp);
    if (!feof(fp)) {
      j = k = 0;
      while (str[k] != '~') temp[j++] = str[k++];
      temp[j] = NULL;
      entry[count].length = (short) atoi(temp);
      k++;
      j = 0;

      entry[count].string = malloc(entry[count].length+1);
      if (!entry[count].string) {
        printf("memory allocation failure\n");
        exit(1);
      }

      while (str[k] != '~') entry[count].string[j++] = str[k++];
      entry[count].string[j] = NULL;
      k++;
      j = 0;

      while (str[k] != '~') temp[j++] = str[k++];
      temp[j] = NULL;
      entry[count].token = (unsigned) atoi(temp);
      k++;
      j = 0;
      while (str[k] != '~') temp[j++] = str[k++];
      temp[j] = NULL;
      entry[count].occurrences = (short) atoi(temp);

      if (index_char)   /* see if new index entry needed */
        if ((this_char=toupper(entry[count].string[0])) > last_char) {
          alpha_index[this_char - 'A'] = count;
          last_char = this_char;
        }
      count++;
    }
  }
  fclose(fp);

  if (!index_char)    /* ie, no index created for "%....%" patterns */
    return 1;
  else
    if ((alpha_index[toupper(*index_char)-'A']) == -1)
      return 0;   /* no entries beginning with first letter of pat */
    else
      return 1;   /* there is, so search can proceed */
}

########################################################################


/* File name : search.c */

#include "stdio.h"
#include "headers.h"

extern   int        alpha_index[];
extern   int        hits;
extern   unsigned   count;

uchar      found, offset;
unsigned   start, stop;

void fixed        /* for patterns with no wildcards in them */
  (uchar *pat, ushort plen)
{
  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {    /* first char in pat is _ character */
    start = 0;
    stop  = count;
  }

  for (; start<stop; start++)
    if (plen == entry[start].length) {
      found = exact(pat, entry[start].string, plen);
      if (found) {
#if ! BIGLOOP
        printf("%s\n", entry[start].string);
#endif
```

```
        hits++;
      }
    }
}

void front  /* for patterns like: "abcde%"  or "%abcde" */
  (uchar *pat, ushort plen, ushort search_type)
{
  uchar rev_pat[MAXSTRING];

  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {
    start = 0;
    stop  = count;
  }

  for (; start<stop; start++)
    if (plen <= entry[start].length) {
      found = exact(pat, entry[start].string, plen);
      if (found) {
        if (search_type==4) {
          strcpy(rev_pat, entry[start].string);
          reverse_pat(rev_pat);
#if ! BIGLOOP
          printf("%s\n", rev_pat);
#endif
        }
        else {
#if ! BIGLOOP
          printf("%s\n", entry[start].string);
#endif
        }
        hits++;
      }
    }
}

void mid          /* for patterns like: "abc%def" */
  (uchar *pat, ushort plen, short wild)
{
  ushort   len;
  uchar    *current;

  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {
    start = 0;
    stop  = count;
  }

  for (; start<stop; start++)
```

```
    if (plen <= entry[start].length) {
      current = entry[start].string;
      found = exact(pat, current, wild);
      if (found) {
        len = entry[start].length;
        found = exact(pat+wild+1, current+(len-(plen-wild)), plen-wild);
        if (found) {
#if ! BIGLOOP
          printf("%s\n", entry[start].string);
#endif
          hits++;
        }
      }
    }
}

void midlast   /* for patterns like: "abc%def%"  & "%abc%def" */
  (uchar *pat, ushort plen, short wild, ushort search_type)
{
  uchar temp_pat[MAXSTRING], rev_pat[MAXSTRING];

  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {
    start = 0;
    stop  = count;
  }

  strcpy(temp_pat, pat);
  temp_pat[plen+1] = NULL;   /* drop last % from pattern  */

  for (; start<stop; start++)
    if (plen <= entry[start].length) {
      found = exact(temp_pat, entry[start].string, wild);
      if (found) {
        found = elastic(temp_pat+wild+1, entry[start].string+wild,
                                                           plen-wild);
        if (found) {
          if (search_type==6) {
            strcpy(rev_pat, entry[start].string);
            reverse_pat(rev_pat);
#if ! BIGLOOP
            printf("%s\n", rev_pat);
#endif
          }
          else
#if ! BIGLOOP
            printf("%s\n", entry[start].string);
#endif
          hits++;
      }
      }
    }
}
```

```
void bothends                    /* for patterns like: "%abcde%"   */
  (uchar *pat, ushort plen)
{
  uchar temp_pat[MAXSTRING];
  unsigned start, found;

  strcpy(temp_pat, pat);
  temp_pat[plen+1] = NULL;          /* drop last '%' from pattern */

  for (start = 0; entry[start].string; start++)
    if (plen <= entry[start].length) {
      found = elastic(temp_pat+1, entry[start].string, plen);
      if (found) {
#if ! BIGLOOP
        printf("%s\n", entry[start].string);
#endif
        hits++;
      }
    }
}

void midmid               /* for patterns like: "abc%def%xyz"   */
  (uchar *pat, ushort plen, short w1, short w2)
{
  uchar      temp, temp_pat[MAXSTRING], *current;
  ushort     nullpos;

  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {
    start = 0;
    stop  = count;
  }

  strcpy(temp_pat, pat);

  for (; start<stop; start++)
    if (plen <= entry[start].length) {
      current = entry[start].string;
      found = exact(temp_pat, current, w1);
      if (found) {                              /* "match%....%...." */
        nullpos = entry[start].length - (plen-w2) - 1;
        found = exact(temp_pat+w2+1, current+nullpos, plen-w2+1);
        if (found) {                            /* "....%....%match" */
          temp = current[nullpos];              /* save char where   */
          current[nullpos] = NULL;              /* null to go.       */
          temp_pat[w2] = NULL;                  /* stop pat at 2nd % */
          found = elastic(temp_pat+w1+1, current+w1, w2-w1-1);
          current[nullpos] = temp;              /* and replace char  */
          if (found) {
#if ! BIGLOOP
            printf("%s\n", entry[start].string);
#endif
            hits++;
          }
        }
      }
    }
}

int elastic                 /* looks for p to appear anywhere in t  */
  (uchar *p, uchar *t, ushort patlen)
{
  short textlen = strlen(t), i = patlen, j = patlen, k;

  while (j > 0 && i <= textlen) {
    k = i;
    while (j > 0 && p[j-1] == toupper(t[k-1]) || p[j-1] == '_') {
      k--;
      j--;
    }
    if (j > 0) {
      i++;
      j = patlen;
    }
    else return 1;
  }
  return 0;
}

int exact                  /* looks for exact match */
  (uchar *p, uchar *t, short i)
{
  while (i > 0 && p[i-1] == toupper(t[i-1]) || p[i-1] == '_') i--;
  return !i ? 1 : 0;
}
```

A.6 'C' program files for text searching using 5.5 million records

```
/* File name : headers.h */
/* Files which use this are: main.c pat.c load.c search.c  */

#define  MAXSTRING      30          /* max length of any string */
#define  MAXLENGTH      5800000     /* max number of entries in table */

typedef  unsigned char   uchar;
typedef  unsigned short  ushort;

/* function prototypes  */

int     get_pattern  (uchar *pattern, ushort *patlen);
int     load         (uchar *index_char, char *file_name);
void    reverse_pat  (uchar *s);
void    fold_index   (void);
void    fixed        (uchar *pat, ushort plen);
void    front        (uchar *pat, ushort plen, ushort search_type);
void    mid          (uchar *pat, ushort plen, short wild);
void    midlast      (uchar *pat, ushort plen, short wild,
                                          ushort search_type);
void    bothends     (uchar *pat, ushort plen);
void    midmid       (uchar *pat, ushort plen, short w1, short w2);
int     exact        (uchar *p, uchar *t, short i);
int     elastic      (uchar *p, uchar *t, ushort patlen);

/***********************************************************************/
/* enumeration used to classify search patterns, thus:   | with:      */
/*                                                        |            */
/* NONE = "........."  L = "........%"  M = "....%...."  | F = first  */
/*   ML = "....%...%"  F = "%........" FL = "%.......%"  | L = last   */
/*   FM = "%...%...." MM = "..%...%.."                   | M = mid    */
/***********************************************************************/

enum pattern_type {NONE, L, M, ML, F, FL, FM, MM};

struct record {              /* used for each entry in the table */
  ushort     length;             /* string length */
  uchar      *string;            /* actual string */
  unsigned   token;              /* id for the string */
  ushort     occurrences;        /* how many triples it appears in */
} entry[MAXLENGTH];

short wild[3];      /* place holders for any '%' wildcards in pattern */
```

```
/* File name : main.c */

#include "stdio.h"
#include "time.h"
#include "head.h"

int alpha_index[27] = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
                      -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, MAXLENGTH};

unsigned long hits;

void main (void)
{
  uchar                pattern[MAXSTRING], *p;
  ushort               patlen, in_index;
  enum pattern_type    search_pat;

  search_pat = get_pattern(pattern, &patlen);

  printf("%ld\n",time(NULL));

  switch (search_pat) {
    case 0: case 1: case 2: case 3: case 7:
      in_index = load(&pattern[0],"forward.txt");
      if (!in_index) {
        printf("Not in table\n");
        exit(1);
      }
      fold_index();
      break;
    case 4: case 6:
      in_index = load(&pattern[patlen-1],"reverse.txt");
      if (!in_index) {
        printf("Not in table\n");
        exit(1);
      }
      reverse_pat(pattern);
      fold_index();
      break;
    case 5:
      load(NULL,"forward.txt");
      break;
  }

  p = pattern;
  while(*p) *p++ = toupper(*p);

  hits = 0;

/*  printf("%ld\n",time(NULL)); */

  switch (search_pat) {
    case 0: fixed(pattern, patlen);
            break;
    case 1: front(pattern, patlen-1, 1);
            break;
    case 2: mid(pattern, patlen-1, wild[0]);
            break;
    case 3: midlast(pattern, patlen-2, wild[0], 3);
            break;
    case 4: front(pattern, patlen-1, 4);
            break;
    case 5: bothends(pattern, patlen-2);
            break;
    case 6: midlast(pattern, patlen-2, patlen - 1 - wild[1], 6);
            break;
    case 7: midmid(pattern, patlen-2, wild[0], wild[1]);
            break;
  }

  printf("%ld\n",time(NULL));

  if (hits) printf(" %d matches found\n", hits);
  else      printf(" Pattern not in table\n");
}

void reverse_pat
  (uchar *s)
{
  uchar c, i = 0, j = strlen(s) - 1;

  for (; i<j; i++, j--) {
    c = s[i];
    s[i] = s[j];
    s[j] = c;
  }
}

void fold_index
  (void)          /* if any index entries are still -1 they */
{                 /* need changing to reflect the start of   */
  short i = 25;         /* the next letter.                    */

  for (; i >= 0; i--)
    if (alpha_index[i] == -1) alpha_index[i] = alpha_index[i+1];
}

#########################################################################

/* File name : pat.c */

#include "stdio.h"
#include "head.h"

int get_pattern
  (uchar  *pattern, ushort *patlen)
                        /* Gets pattern > 3 characters long with  */
                        /* 0, 1 or 2 wildcards in it.  Returns to */
  uchar  i, total;      /* "patlen", and the enumerated integer   */
  uchar  ok = 0;        /* for the search strategy to be used.    */
  uchar  c, *w;

  do {              /* continue until we have a pattern which is ok */
    w = pattern;
    printf("enter pattern length 4 or greater : ");
    while (isspace(c=getchar()))
```

```
      ;                 /* skip leading while space */
    *w++ = c;
    while ((*w = getchar()) != '\n')  /* now get pattern until CR hit */
      w++;
    *w = NULL;           /* append null byte */
    *patlen = strlen(pattern);   /* and get length */
    if (*patlen < 4) printf("too small - re-enter\n");
    else {
      wild[0] = wild[1] = wild[2] = -1;/* holds positions of wildcards*/
      for (i=0, total=0; total<3 && pattern[i]; i++)
        if (pattern[i] == '%') wild[total++] = i;
      switch (total) {  /* total now = number of wildcards in pattern */
        case 0: return NONE;
        case 1: ok = 1;
                break;
        case 2: if (wild[1] - wild[0] == 1)
                                    /* adjacent %s + pattern short*/
                  if (*patlen == 4)
                    printf("short pattern - re-enter\n");
                  else {             /* remove one of the wildcards */
                    for (i=wild[1]; pattern[i]; i++)
                        pattern[i] = pattern[i+1];
                    (*patlen)--;
                    wild[1] = -1;
                    ok = 1;
                  }
                else ok = 1;
                break;
        case 3: printf("too many wildcards - try again\n");
                break;
      }
    }
  } while (!ok); /* pattern has zero, one or two wildcards only in it */

  if (wild[1] == -1) {         /* ie, only one wildcard in pattern */
    if (wild[0] == *patlen-1) return L;
    if (wild[0] > 0) return M;
    else return F;
  }
  else                                /* two wildcards in pattern */
    if (wild[1] == *patlen-1)
      if (wild[0] > 0) return ML;
      else return FL;
    else if (wild[0] == 0) return FM;
         else return MM;
}
```

```
/* File name : load.c */

#include "stdio.h"
#include "head.h"
#include "stdlib.h"

extern int alpha_index[];
unsigned count = 0;

int load
  (uchar *index_char, char *file_name)
{
  FILE *fp;
  char str[100], temp[20];
  uchar last_char = '@', this_char;
  unsigned j, k;

  if ((fp = fopen(file_name,"r"))==NULL) {
    printf("cannot open in file\n");
    exit(1);
  }

  count = 0;

  while (!feof(fp)) {
    fgets(str, 100, fp);
    if (!feof(fp)) {
      j = k = 0;
      while (str[k] != '~') temp[j++] = str[k++];
      temp[j] = NULL;
      entry[count].length = (short) atoi(temp);
      k++;
      j = 0;

      entry[count].string = malloc(entry[count].length+1);
      if (!entry[count].string) {
        printf("memory allocation failure\n");
        exit(1);
      }

      while (str[k] != '~') entry[count].string[j++] = str[k++];
      entry[count].string[j] = NULL;
      k++;
      j = 0;

      while (str[k] != '~') temp[j++] = str[k++];
      temp[j] = NULL;
      entry[count].token = (unsigned) atoi(temp);
      k++;
      j = 0;

      while (str[k] != '~') temp[j++] = str[k++];
      temp[j] = NULL;
      entry[count].occurrences = (short) atoi(temp);
      if (index_char)   /* see if new index entry needed */
        if ((this_char=toupper(entry[count].string[0])) > last_char) {
          alpha_index[this_char - 'A'] = count;
          last_char = this_char;
        }
      count++;
    }
  }
  fclose(fp);

  if (!index_char)    /* ie, no index created for "%....%" patterns */
    return 1;
  else
    if ((alpha_index[toupper(*index_char)-'A']) == -1)
      return 0;   /* no entries beginning with first letter of pat */
    else
      return 1;   /* there is, so search can proceed */
}

########################################################################

/* File name : search.c */

#include "stdio.h"
#include "head.h"

extern   int        alpha_index[];
extern   int        hits;
extern   unsigned   count;

uchar      found, offset;
unsigned   start, stop;

void fixed         /* for patterns with no wildcards in them */
  (uchar *pat, ushort plen)
{
  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {    /* first char in pat is _ character */
    start = 0;
    stop  = count;
  }

  for (; start<stop; start++)
    if (plen == entry[start].length) {
      found = exact(pat, entry[start].string, plen);
      if (found) hits++;
    }
}

void front  /* for patterns like: "abcde%"  or "%abcde" */
  (uchar *pat, ushort plen, ushort search_type)
{
  uchar rev_pat[MAXSTRING];
  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {
    start = 0;
    stop  = count;
  }

  for (; start<stop; start++)
    if (plen <= entry[start].length) {
      found = exact(pat, entry[start].string, plen);
      if (found) {
        if (search_type==4) {
          strcpy(rev_pat, entry[start].string);
          reverse_pat(rev_pat);
        }
        hits++;
      }
    }
}

void mid          /* for patterns like: "abc%def" */
  (uchar *pat, ushort plen, short wild)
{
  ushort   len;
  uchar    *current;

  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {
    start = 0;
    stop  = count;
  }

  for (; start<stop; start++)
    if (plen <= entry[start].length) {
      current = entry[start].string;
      found = exact(pat, current, wild);
      if (found) {
        len = entry[start].length;
        found = exact(pat+wild+1, current+(len-(plen-wild)), plen-wild);
        if (found) hits++;
      }
    }
}

void midlast   /* for patterns like: "abc%def%"  & "%abc%def" */
  (uchar *pat, ushort plen, short wild, ushort search_type)
{
  uchar temp_pat[MAXSTRING], rev_pat[MAXSTRING];

  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {
    start = 0;
    stop  = count;
  }

  strcpy(temp_pat, pat);
  temp_pat[plen+1] = NULL;   /* drop last % from pattern  */

  for (; start<stop; start++)
    if (plen <= entry[start].length) {
      found = exact(temp_pat, entry[start].string, wild);
      if (found) {
        found = elastic(temp_pat+wild+1, entry[start].string+wild,
                                                          plen-wild);
        if (found) {
          if (search_type==6) {
            strcpy(rev_pat, entry[start].string);
            reverse_pat(rev_pat);
          }
          else hits++;
        }
      }
    }
}

void bothends                 /* for patterns like: "%abcde%"  */
  (uchar *pat, ushort plen)
{
  uchar temp_pat[MAXSTRING];
  unsigned start, found;

  strcpy(temp_pat, pat);
  temp_pat[plen+1] = NULL;         /* drop last '%' from pattern */

  for (start = 0; entry[start].string; start++)
    if (plen <= entry[start].length) {
      found = elastic(temp_pat+1, entry[start].string, plen);
      if (found) hits++;
    }
}

void midmid             /* for patterns like: "abc%def%xyz"  */
  (uchar *pat, ushort plen, short w1, short w2)
{
  uchar      temp, temp_pat[MAXSTRING], *current;
  ushort     nullpos;

  if (isalpha(pat[0])) {
    offset = pat[0]-'A';
    start  = alpha_index[offset];
    stop   = alpha_index[offset+1];
  }
  else {
    start = 0;
    stop  = count;
  }

  strcpy(temp_pat, pat);

  for (; start<stop; start++)
    if (plen <= entry[start].length) {
      current = entry[start].string;
      found = exact(temp_pat, current, w1);
      if (found) {                              /* "match%....%...." */
        nullpos = entry[start].length - (plen-w2) - 1;
        found = exact(temp_pat+w2+1, current+nullpos, plen-w2+1);
        if (found) {                            /* "....%....%match" */
          temp = current[nullpos];              /* save char where   */
          current[nullpos] = NULL;              /* null to go.       */
          temp_pat[w2] = NULL;                  /* stop pat at 2nd % */
          found = elastic(temp_pat+w1+1, current+w1, w2-w1-1);
          current[nullpos] = temp;              /* and replace char  */
          if (found) hits++;
        }
      }
    }
}

int elastic                /* looks for p to appear anywhere in t  */
  (uchar *p, uchar *t, ushort patlen)
{
  short textlen = strlen(t), i = patlen, j = patlen, k;

  while (j > 0 && i <= textlen) {
    k = i;
    while (j > 0 && p[j-1] == toupper(t[k-1]) || p[j-1] == '_') {
      k--;
      j--;
    }
    if (j > 0) {
      i++;
      j = patlen;
    }
    else return 1;
  }
  return 0;
}

int exact                  /* looks for exact match */
  (uchar *p, uchar *t, short i)
{
  while (i > 0 && p[i-1] == toupper(t[i-1]) || p[i-1] == '_') i--;
  return !i ? 1 : 0;
}
```

A.7 Initial functions used for experimental searches

```
/*********************************/
/* Program new_strings.c        */
/* includes object functions for */
/* string matching.             */
/* written by S. Sheldrake 1999  */
/*********************************/
#include <stdio.h>
#include <setjmp.h>
#include "def.h"

#define ALPHA_SIZE 256
#define DELIMITER " "

char text[MAXSTRLEN];
char pat[MAXSTRLEN];
char subpat[MAXSTRLEN];
char word[MAXSTRLEN];

int em1(char *t, char *p, int s_type)
/* searches for pat in text using a one-at-a-time move strategy */
/* looks for exact word matches only using #define DELIMITER */
{
  strcpy(text,DELIMITER);        /* Add delimiter */
  strcpy(pat,DELIMITER);         /* to font and   */
  strcat(text,t);                /* back of text  */
  strcat(pat,p);                 /* and pattern   */
  strcat(text,DELIMITER);
  strcat(pat,DELIMITER);

  if (s_type == 0)
    return exact(text,pat,strlen(text)) >= 0 ? (OK) : (FAIL);
  else
    return shift(text,pat,strlen(text)) >= 0 ? (OK) : (FAIL);
}

int em2(char *t, char *p)
/* searches for pat in text using a one-at-a-time */
/* strategy and permits missing characters in pat */
/* looks for delimited, exact word matches only   */
{
  int i, j, k, l_text, patlen;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);
  strcat(pat,p);
  strcat(text,DELIMITER);
  strcat(pat,DELIMITER);

  i = j = patlen = strlen(pat);
  l_text = strlen(text);

  while (j > 0 && i <= l_text) {
    k = i;
    while (j > 0 && pat[j-1] == text[k-1] || pat[j-1] == '_') {
      j--;
      k--;
    }
    if (j > 0) {
      i++;
      j = patlen;
    }
    else return (OK);
  }
  return (FAIL);
}

int em21(char *t, char *p)
/* searches for pat in text using a shift       */
/* table and permits missing characters in pat  */
/* looks for delimited, exact word matches only */
{
  int i, j, k, l_text, patlen, shift, table[ALPHA_SIZE];
  char *ptr;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);              /* Add DELIMITER */
  strcat(pat,p);               /* to front and  */
  strcat(text,DELIMITER);      /* back of text  */
  strcat(pat,DELIMITER);       /* and pattern   */

  patlen = strlen(pat);
  ptr = pat;

  for (i=patlen -1, shift = -1; i > -1 && shift < 0; i--)
    if (pat[i] == '_') shift = patlen - i - 1;
  if (shift == 0) shift++;
  if (shift == -1) shift = patlen;
  for (i=0; i<ALPHA_SIZE; i++) table[i] = shift;
  for (i=1; i<patlen; i++, ptr++) table[*ptr] = patlen - i;

  i = j = patlen;
  l_text = strlen(text);

  while (j > 0 && i <= l_text) {
    k = i;
    while (j > 0 && pat[j-1] == text[k-1] || pat[j-1] == '_') {
      j--;
      k--;
    }
    if (j > 0) {
      i += table[text[i-1]] < shift ? table[text[i-1]] : shift;
      j = patlen;
    }
    else return (OK);
  }
  return (FAIL);
}

int em3(char *t, char *p)
/* searches for pat in text using a one-at-a-time move strategy */
/* copes with right-hand truncation by discarding wildcard char  */
/* looks for exact word matches only using #define DELIMITER    */
{
  int i, j, k, l_text, patlen;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);            /* Add DELIMITER to front and end of */
  strcat(text,DELIMITER);    /* text and to front of pattern.     */
  strcat(pat,p);

  patlen = strlen(pat);
  patlen--;                  /* gets rid of wild card character   */
  pat[patlen]=NULL;          /* and null terminate pat.           */
  l_text = strlen(text);
  i = j = 0;                 /* this time search is from front    */

  while (j < patlen && i <= l_text) {
    k = i;
    while (j < patlen && pat[j] == text[k]) {
      j++;
      k++;
    }
    if (j < patlen) {
      i++;
      j = 0;
    }
    else return (OK);
  }
  return (FAIL);
}

int em4(char *t, char *p, int s_type)
/* searches for pat in text using a one-at-a-time move strategy */
/* or using a shift table depending on value of s_type.         */
/* Copes with left-hand truncation by discarding wildcard char  */
/* looks for exact word matches only using #define DELIMITER    */
{
  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);            /* Add DELIMITER to front and end of */
  strcat(text,DELIMITER);    /* text and to end only of pattern.  */
  strcpy(pat,p+1);           /* when copying p to pat, make sure  */
  strcat(pat,DELIMITER);     /* to dump the wildcard character    */

  if (s_type == 0)
    return exact(text,pat,strlen(text)) >= 0 ? (OK) : (FAIL);
  else
    return shift(text,pat,strlen(text)) >= 0 ? (OK) : (FAIL);
}

int em5(char *t, char *p)
/* searches for pat in text using a one-at-a-time move strategy */
/* copes with wildcard char anywhere in middle of search pat    */
/* looks for exact word matches only using #define DELIMITER    */
{
  int i, j, k, l_text, patlen, subpatlen;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);               /* Add DELIMITER to front and end of */
  strcat(text,DELIMITER);       /* text and pattern.  */
  strcat(pat,p);
  strcat(pat,DELIMITER);

  patlen = strlen(pat);
  patlen--;                    /* gets rid of wildcard to be ignored */
  i = j = patlen;
  l_text = strlen(text);

  while (j > 0 && i <= l_text) {
    k = i;
    while (j > 0 && text[k] == pat[j]) {       /* find RH end first */
      j--;
      k--;
    }
    if (pat[j] == '%') {        /* set things up to look from LH end */
      strcpy(subpat,pat);       /* first make copy of pat */
      subpat[j]=NULL;           /* and shorten it to LH end bit only */
      while (text[k] != ' ') k--;       /* line up k to be at */
      subpatlen = j;                          /* beginning of word  */
      j = 0;               /* store length of sub pat and set j to 0 */
      while (j < subpatlen && pat[j] == text[k]) {
        j++;                  /* now do search from LH end to either */
        k++;                  /* mismatch or success for sub pat     */
      }
      if (j < subpatlen) {
        i++;
        j = patlen;
      }
      else return (OK);
    } /* endif pat[j] == '%' */
    else {
      if (j > 0) {
        i++;
        j = patlen;
      }
      else return (OK);
    }
  } /* while */
  return (FAIL);
}

int em6(char *text, char *pat, int s_type)
/* takes pat, which now has an elastic meta character at each */
/* end, lops off the meta characters and does one-at-a-time   */
/* search or using shift table (depends on s_type passed in.) */
{
  pat++;                         /* lose first % character     */
  pat[strlen(pat)-1] = NULL;     /* lose last % character      */

  if (s_type == 0)
    return exact(text,pat,strlen(text)) >= 0 ? (OK) : (FAIL);
  else
    return shift(text,pat,strlen(text)) >= 0 ? (OK) : (FAIL);
}
```

```
int mm1(char *t, char *pat)
/* takes pat, which is a multiple pattern of patterns separated by */
/* | (bang) character, and searches for the sub-patterns - any of  */
/* which will result in success being returned.  Exact words only. */
{
  char *ptr;
  int match, l_text;

  strcpy(text,DELIMITER);
  strcpy(subpat,DELIMITER);
  strcat(text,t);              /* adds the DELIMITER to either */
  strcat(text,DELIMITER);      /* end of the textsstring       */
  l_text = strlen(text);

  while (pat) {
    ptr = subpat;
    ptr++;
    while ((*ptr++ = *pat++) != '|' && *pat != NULL)
      ;
    if (*(ptr-1) == '|') *--ptr = ' ';      /* these lines add  */
    else *ptr = ' ';              /* on the final DELIMITER and  */
    *++ptr = NULL;                              /* the NULL byte */
    match = exact(text,subpat,l_text);
    if (match>=0) return (OK);        /* found match - so return ok   */
    if (*pat == '|') pat++;
    if (*pat == NULL) return (FAIL);
  }
}

int mm2(char *t, char *pat)
/* takes pat, which is a multiple pattern of patterns separated by  */
/* & (ampersand) character, and searches for the sub-patterns - all */
/* of which must be in text for success.  Exact words only.         */
{
  char *ptr;
  int match, l_text;

  strcpy(text,DELIMITER);
  strcpy(subpat,DELIMITER);
  strcat(text,t);              /* adds the DELIMITER to either */
  strcat(text,DELIMITER);      /* end of the textsstring       */
  l_text = strlen(text);

  while (pat) {          /* there's still a subpat to process */
    ptr = subpat;        /* use ptr to move thru' subpat      */
    ptr++;               /* skip leading DELIMITER            */
    while ((*ptr++ = *pat++) != '&' && *pat != NULL)
      ;
    if (*(ptr-1) == '&') *--ptr = ' ';      /* these lines add  */
    else *ptr = ' ';              /* on the final DELIMITER and  */
    *++ptr = NULL;                              /* the NULL byte */
    match = exact(text,subpat,l_text);
    if (match<0) return (FAIL);         /* if any supbats not in text */
    if (*pat == '&') pat++;                  /* return fail */
    if (*pat == NULL) return (OK);
  }
}
```

```
int mm3(char *t, char *pat)
/* takes pat, which is a multiple pattern of patterns separated by   */
/* < (less than) character, and searches for the sub-patterns - all  */
/* of which must be in text and in ascending order.  Exact word only */
{
  char *ptr;
  int match, last_match = 0, l_text;

  strcpy(text,DELIMITER);
  strcpy(subpat,DELIMITER);
  strcat(text,t);               /* adds the DELIMITER to either */
  strcat(text,DELIMITER);       /* end of the text string       */
  l_text = strlen(text);

  while (pat) {            /* there's still a subpat to process */
    ptr = subpat;          /* use ptr to move thru' subpat      */
    ptr++;                 /* skip leading DELIMITER            */
    while ((*ptr++ = *pat++) != '<' && *pat != NULL)
      ;
    if (*(ptr-1) == '<') *--ptr = ' ';      /* these lines add  */
    else *ptr = ' ';                /* on the final DELIMITER and  */
    *++ptr = NULL;                               /* the NULL byte */
    match = exact(text,subpat,l_text);
    if (match < 0 || match < last_match) return (FAIL);
    last_match = match;             /* make sure matches are found */
    if (*pat == '<') pat++;         /* in the correct ordering     */
    if (*pat == NULL) return (OK);
  }
}

int exact(char *text, char *pat, int l_text)
/* uses one-at-a-time shift to find match */
{
  int i, j, k, patlen = strlen(pat);

  i = j = patlen;

  while (j > 0 && i <= l_text) {
    k = i;
    while (j > 0 && text[k-1] == pat[j-1]){
      j--;
      k--;
    }
    if (j > 0) {
      i++;
      j = patlen;
    }
    else return k;
  }
  return -1;
}

int shift(char *text, char *pat, int l_text)
/* uses shift table to find exact match */
{
  int i, j, k, patlen = strlen(pat), table[ALPHA_SIZE];
  char *ptr = pat;
  for (i=0; i<ALPHA_SIZE; i++) table[i] = patlen;
  for (i=1; i<patlen; i++, ptr++) table[*ptr] = patlen-i;

  i = j = patlen;

  while (j > 0 && i <= l_text) {
    k = i;
    while (j > 0 && text[k-1] == pat[j-1]){
      j--;
      k--;
    }
    if (j > 0) {
      i += table[text[i-1]];
      j = patlen;
    }
    else return k;
  }
  return -1;
}

int ss1(char *t, char *p)
/* searches for pat in text using on-at-a-time moves */
/* returns the number of times pat appears in text   */
{
  int l_text, patlen, i, j, k, count = 0;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);
  strcat(text,DELIMITER);
  strcat(pat,p);
  strcat(pat,DELIMITER);

  l_text = strlen(text);
  patlen = strlen(pat);
   i = j = patlen;

  while (i <= l_text) {
    k = i;
    while (j > 0 && text[k-1] == pat[j-1]) {
      k--;
      j--;
    }
    if (j < 1) count++;
    i++;
    j = patlen;
  }
  return count;
}

int ss2(char *t, char *p)
/* searches for pat in text using shift table and  */
/* returns the number of times pat appears in text */
{
  char *ptr;
  int l_text, patlen, table[ALPHA_SIZE], i, j, k, count = 0;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);
  strcat(text,DELIMITER);
  strcat(pat,p);
  strcat(pat,DELIMITER);
  l_text = strlen(text);
  patlen = strlen(pat);
  ptr = pat;

  for (i = 0; i < ALPHA_SIZE; i++) table[i] = patlen;
  for (i = 1; i < patlen; i++, ptr++) table[*ptr] = patlen - i;

  i = j = patlen;

  while (i <= l_text) {
    k = i;
    while (j > 0 && text[k-1] == pat[j-1]) {
      k--;
      j--;
    }
    if (j < 1) count++;
    i += table[text[i-1]];
    j = patlen;
  }
  return count;
}

char *mm11(char *t, char *p)
/* searches for pat in text using a one-at-a-time move strategy */
/* looks for exact word matches only using #define DELIMITER */
/* returns string found or empty string if not found */
{
  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);                /* Add DELIMITER */
  strcat(pat,p);                 /* to front and  */
  strcat(text,DELIMITER);        /* back of text  */
  strcat(pat,DELIMITER);         /* and pattern   */

  return exact(text,pat,strlen(text)) >= 0 ? (p) : ("");
}

char *mm22(char *t, char *p)
/* searches for pat in text using a one-at-a-time */
/* strategy and permits missing characters in pat */
/* looks for delimited, exact word matches only   */
/* returns the word from the text that caused match */
{
  int i, j, k, l_text, patlen;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);                /* Add DELIMITER */
  strcat(pat,p);                 /* to front and  */
  strcat(text,DELIMITER);        /* back of text  */
  strcat(pat,DELIMITER);         /* and pattern   */

  i = j = patlen = strlen(pat);
  l_text = strlen(text);

  while (j > 0 && i <= l_text) {
    k = i;
    while (j > 0 && pat[j-1] == text[k-1] || pat[j-1] == '_') {
      j--;
      k--;
    }
    if (j > 0) {
      i++;
      j = patlen;
    }
    else {
      strncpy(word, text+k+1, i-k-1);
      word[i-k-2] = NULL;
      return word;
    }
  }
  return "";
}

char *mm33(char *t, char *p)
/* searches for pat in text using a one-at-a-time move strategy */
/* copes with right-hand truncation by discarding wildcard char */
/* looks for exact word matches only using #define DELIMITER    */
/* returns the word in the text that made the match */
{
  int i, j, k, l_text, m, patlen;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);             /* Add DELIMITER to front and end of */
  strcat(text,DELIMITER);     /* text and to front of pattern.     */
  strcat(pat,p);

  patlen = strlen(pat);
  patlen--;                   /* gets rid of wild card character   */
  pat[patlen]=NULL;           /* and null terminate pat.           */
  l_text = strlen(text);
  i = j = 0;                  /* this time search is from front    */

  while (j < patlen && i <= l_text) {
    k = i;
    while (j < patlen && pat[j] == text[k]) {
      j++;
      k++;
    }
    if (j < patlen) {
      i++;
      j = 0;
    }
    else {
      m = 0;       /* to step through word */
      i++;         /* move i past first space char in text */
      while ((word[m++] = text[i++]) != ' ')
        ;
      word[m] = NULL;
      return word;
    }
  }
  return "";
}

char *mm44(char *t, char *p)
/* searches for pat in text using a one-at-a-time move strategy */
/* Copes with left-hand truncation by discarding wildcard char  */
/* looks for exact word matches only using #define DELIMITER    */
/* returns the word in the text that made the match  */
{
  int n, m = 0;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);             /* Add DELIMITER to front and end of */
  strcat(text,DELIMITER);     /* text and to end only of pattern.  */
  strcpy(pat,p+1);            /* when copying p to pat, make sure  */
  strcat(pat,DELIMITER);      /* to dump the wildcard character    */

  n = exact(text,pat,strlen(text));
  if (n >=0) {
    while (text[n] != ' ') n--; /* gets n to beginning of match word */
    while ((word[m++] = text[++n]) != ' ')
      ;
    word[m] = NULL;
    return word;
  }
  else return "";
}

char *mm55(char *t, char *p)
/* searches for pat in text using a one-at-a-time move strategy */
/* copes with wildcard char anywhere in middle of search pat    */
/* looks for exact word matches only using #define DELIMITER    */
/* returns the word in text that made the match */
{
  int i, j, k, l_text, patlen, subpatlen, m = 0, n;

  strcpy(text,DELIMITER);
  strcpy(pat,DELIMITER);
  strcat(text,t);                /* Add DELIMITER to front and end of */
  strcat(text,DELIMITER);        /* text and pattern.  */
  strcat(pat,p);
  strcat(pat,DELIMITER);

  patlen = strlen(pat);
  patlen--;                     /* gets rid of wildcard to be ignored */
  i = j = patlen;
  l_text = strlen(text);

  while (j > 0 && i <= l_text) {
    n = k = i;
    while (j > 0 && text[k] == pat[j]) {        /* find RH end first */
      j--;
      k--;
    }
    if (pat[j] == '%') {         /* set things up to look from LH end */
```

```
      strcpy(subpat,pat);       /* first make copy of pat */
      subpat[j]=NULL;           /* and shorten it to LH end bit only */
      while (text[k] != ' ') k--;       /* line up k to be at */
      subpatlen = j;                          /* beginning of word  */
      j = 0;                 /* store length of sub pat and set j to 0 */
      while (j < subpatlen && pat[j] == text[k]) {
        j++;                  /* now do search from LH end to either */
        k++;                  /* mismatch or success for sub pat     */
      }
      if (j < subpatlen) {
        i++;
        j = patlen;
      }
      else {
        n--;
        while (text[n] != ' ')
          n--;                 /* gets n to beginning of match word */
        while ((word[m++] = text[++n]) != ' ')
          ;
        word[m] = NULL;
        return word;
      }
    }                         /* endif pat[j] == '%' */
    else {
      if (j > 0) {
        i++;
        j = patlen;
      }
      else {
        n--;
        while (text[n] != ' ')
          n--;                /* gets n to beginning of match word */
        while ((word[m++] = text[++n]) != ' ')
          ;
        word[m] = NULL;
        return word;
      }
    }
  } /* while */
  return "";
}

char *mm66(char *t, char *pat)
/* takes pat, which is a multiple pattern of patterns separated by */
/* | (bang) character, and searches for the sub-patterns - any of  */
/* which will result in success being returned.  Exact words only. */
/* the word in the text that makes the match is returned. */
{
  char *ptr;
  int match, l_text;

  strcpy(text,DELIMITER);
  strcat(text,t);             /* adds the DELIMITER to either */
  strcat(text,DELIMITER);     /* end of the textsstring       */
  strcpy(subpat,DELIMITER);
  l_text = strlen(text);
  while (pat) {
    ptr = subpat;
    ptr++;
    while ((*ptr++ = *pat++) != '|' && *pat != NULL)
      ;
    if (*(ptr-1) == '|') *--ptr = ' ';      /* these lines add  */
    else *ptr = ' ';             /* on the final DELIMITER and  */
    *++ptr = NULL;                              /* the NULL byte */
    match = exact(text,subpat,l_text);
    if (match>=0) {
        subpat[strlen(subpat)-1] = NULL;
        return subpat+1;
    }
    if (*pat == '|') pat++;
    if (*pat == NULL) return "";
  }
}
```

A.8 Final search functions used in software

```
/*********************************/
/* Program new_strings.c         */
/* includes object functions for */
/* string matching.              */
/* written by S. Sheldrake 2000  */
/*********************************/
#include <stdio.h>
#include <setjmp.h>
#include "def.h"

#define DELIMITER ' '

char *word(), *wild(), *embed();
char curr_word[MAXSTRLEN];
char curr_pat[MAXSTRLEN];

char *matches(char *text, char *pat)
/* examines pat to see how many wild cards in it.  If there  */
/* are zero, one or two, it passes pat & text to appropriate */
/* function - either word(), wild() or embed()               */
{
  int i, total, len = strlen(pat);

  for (i = 0, total = 0; total<2 && pat[i]; i++)
    if (pat[i] == '%') total++;

  switch (total) {
    case 0  : return word(text,pat);
    case 1  : return wild(text,pat);
    case 2  : if (pat[0] == '%' && pat[len-1] == '%')
                return embed(text,pat);
              else
                return "incorrect position of wild cards";
    default : return "too many wild cards used";
  }
}

char *word(char *text, char *pat)
/* searches for pat in text using a one-at-a-time move   */
/* allowing for '_' in pat meaning any one character.    */
/* returns first word that matches.                      */
{
  int i, j = 0, x, plen = strlen(pat);

  while (text[j]) {
    i = 0;      /* used as count through each word */
    while ((curr_word[i++] = text[j++]) != DELIMITER && text[j] != NULL)
      ;
    if (text[j-1] == DELIMITER) curr_word[i-1] = NULL;
    else curr_word[i] = NULL;

    x = strlen(curr_word);
    if (plen == x) {
      while (x > 0 && pat[x-1] == curr_word[x-1] || pat[x-1] == '_')
        x--;
      if (!x) return curr_word;
    }
  }
  return "";
}

char *wild(char *text, char *pat)
/* Searches for pat in text.  Allows for one '%' character  */
/* at either end of pat or in middle.  Also allows for '_'  */
/* anywhere in pat.  Returns first word that matches        */
{
  int i, j = 0, x, w = 0, plen = strlen(pat)-1, wlen;
  char *p, *t;

  while (pat[w] != '%' && pat[w] != NULL) w++;

  while (text[j]) {
    i = 0;      /* used as count through each word */
    while ((curr_word[i++] = text[j++]) != DELIMITER && text[j] != NULL)
      ;
    if (text[j-1] == DELIMITER) curr_word[i-1] = NULL;
    else curr_word[i] = NULL;

    wlen = strlen(curr_word);
    if (plen <= wlen) {
      x = w;        /* look for back bit of pat in curr_word */
      while (x > 0 && pat[x-1] == curr_word[x-1] || pat[x-1] == '_')
        x--;
      if (!x) {     /* if found, set up and look for front bit */
        x = plen - w;
        p = pat+w+1;
        t = curr_word+(wlen-(plen-w));
        while (x > 0 && p[x-1] == t[x-1] || p[x-1] == '_') x--;
        if (!x) return curr_word;
      }
    }
  }
  return "";
}

char *embed(char *text, char *pat)
/* takes pat, which now has form '%xxxx%'
/* and returns first word that matches */
{
  int i, j = 0, x, y, z, plen, wlen;

  pat++;                          /* lose first % character     */
  pat[strlen(pat)-1] = NULL;      /* lose last % character      */
  plen = strlen(pat);

  while (text[j]) {
    i = 0;      /* used as count through each word */
    while ((curr_word[i++] = text[j++]) != DELIMITER && text[j] != NULL)
      ;
    if (text[j-1] == DELIMITER) curr_word[i-1] = NULL;
    else curr_word[i] = NULL;

    wlen = strlen(curr_word);
    if (plen <= wlen) {
      x = y = plen;
      while (y > 0 && x <= wlen) {
        z = x;
        while (y > 0 && pat[y-1] == curr_word[z-1] || pat[y-1] == '_') {
          z--;
          y--;
        }
        if (y > 0) {
          x++;
          y = plen;
        }
        else return curr_word;
      }
    }
  }
  return "";
}

char *rest(char *text, char *pat)
/* searches for pat in text using a one-at-a-time move   */
/* returns the rest of the text after the match.          */
{
  int i, j = 0;

  while (text[j]) {
    i = 0;      /* used as count through each word */
    while ((curr_word[i++] = text[j++]) != DELIMITER && text[j] != NULL)
      ;
    if (text[j-1] == DELIMITER) curr_word[i-1] = NULL;
    else curr_word[i] = NULL;

    if (!strcmp(curr_word,pat)) return text+j;
  }
  return "";
}

char *or_str(char *text, char *pat)
/* takes pat, which is a multiple pattern of patterns separated by */
/* | (bang) character, and searches for the sub-patterns - any of  */
/* which will result in that word being returned.                  */
{
  int i, j, y = 0;

  while (pat[y]) {
    i = 0;      /* used as count through each bit of pat */
    while ((curr_pat[i++] = pat[y++]) != '|' && pat[y] != NULL)
      ;
    if (pat[y-1] == '|') curr_pat[i-1] = NULL;
    else curr_pat[i] = NULL;

    j = 0;
    while (text[j]) {
      i = 0;      /* used as count through each word */
      while ((curr_word[i++]=text[j++])!= DELIMITER && text[j] != NULL)
        ;
      if (text[j-1] == DELIMITER) curr_word[i-1] = NULL;
      else curr_word[i] = NULL;
      if (!strcmp(curr_pat,curr_word)) return curr_word;
    }
  }
  if (pat[y] == '|') y++;
  if (pat[y] == NULL) return "";
}

int and_str(char *text, char *pat)
/* takes pat, which is a multiple pattern of patterns separated by  */
/* & (ampersand) character, and searches for the sub-patterns - all */
/* of which must be in text for success.  Exact words only.         */
{
  int i, j, y = 0, found;

  while (pat[y]) {
    i = 0;
    while ((curr_pat[i++] = pat[y++]) != '&' && pat[y] != NULL)
      ;
    if (pat[y-1] == '&') curr_pat[i-1] = NULL;
    else curr_pat[i] = NULL;

    found = 0;
    j = 0;      /* used as count through text */

    while (text[j] && !found) {
      i = 0;      /* used as count through each word */
      while ((curr_word[i++]=text[j++]) != DELIMITER && text[j] != NULL)
        ;
      if (text[j-1] == DELIMITER) curr_word[i-1] = NULL;
      else curr_word[i] = NULL;

      if (!strcmp(curr_word,curr_pat)) found = 1;
    }
    if (text[j] == NULL && !found) return (FAIL);
    if (pat[y] == '&') y++;
    if (pat[y] == NULL) return (OK);
  }
}

int order_str(char *text, char *pat)
/* takes pat, which is a multiple pattern of patterns separated by   */
/* < (less than) character, and searches for the sub-patterns - all  */
/* of which must be in text and in ascending order.  Exact word only */
{
  int i, j = 0, y = 0, found;

  while (pat[y]) {
    i = 0;      /* used as count through bits of pat */
    while ((curr_pat[i++] = pat[y++]) != '<' && pat[y] != NULL)
      ;
    if (pat[y-1] == '<') curr_pat[i-1] = NULL;
    else curr_pat[i] = NULL;

    found = 0;
    while (text[j] && !found) {
      i = 0;      /* used as count through each word */
      while ((curr_word[i++]=text[j++]) != DELIMITER && text[j] != NULL)
```

```
        ;
      if (text[j-1] == DELIMITER) curr_word[i-1] = NULL;
      else curr_word[i] = NULL;

      if (!strcmp(curr_pat,curr_word)) found = 1;
    }
    if (!found) return (FAIL);
    if (text[j] == NULL && pat[y] != NULL) return (FAIL);
    if (pat[y] == '<') y++;
    if (pat[y] == NULL) return (OK);
  }
}

int count_str(char *text, char *pat)
/* returns the number of times pat appears in text */
{
  int i, j = 0, count = 0;

  while (text[j]) {
    i = 0;      /* used as count through each word */
    while ((curr_word[i++] = text[j++]) != DELIMITER && text[j] != NULL)
      ;
    if (text[j-1] == DELIMITER) curr_word[i-1] = NULL;
    else curr_word[i] = NULL;

    if (!strcmp(curr_word,pat)) count++;
  }
  return count;
}

int count_words(char *text, char *pat)
/* returns the number of words in a text   */
{
  int j = 0, count = 0;

  if (text[j]==NULL) return count;

  while (text[j]) {
    while (text[j++] != DELIMITER && text[j] != NULL)
      ;
    if (text[j-1] == DELIMITER && text[j] != NULL) count++;
    if (text[j-2] == '\\') count--;
  }
  return ++count;
}

int exact(char *text, char *pat, int l_text)
/* uses one-at-a-time shift to find match */
{
  int i, j, k, patlen = strlen(pat);

  i = j = patlen;

  while (j > 0 && i <= l_text) {
    k = i;
    while (j > 0 && text[k-1] == pat[j-1]){
      j--;
      k--;
    }
    if (j > 0) {
      i++;
      j = patlen;
    }
    else return k;
  }
  return -1;
}
```