UNIVERSITY OF SOUTHAMPTON

# Optimizing Data Storage for Performance in a Modern Binary Relational Database based on a Triple Store

**by**

**Stephen J O'Connell, MA**

Department of Electronics and Computer Science,
University of Southampton
Southampton, SO17 1BJ
Email: soc@ecs.soton.ac.uk

**December 2002**

Thesis submitted by Stephen O'Connell
in candidature for the degree of
Master of Philosophy
at the University of Southampton

---

Supervisors: Prof A J G Hey, Dr D A Nicole
and N Winterbottom

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE

ELECTRONICS AND COMPUTER SCIENCE

Master of Philosophy

OPTIMIZING DATA STORAGE FOR PERFORMANCE IN A MODERN BINARY RELATIONAL DATABASE BASED ON A TRIPLE STORE

by Stephen John O'Connell

This thesis introduces a new approach to understanding the issues relating to the efficient implementation of a binary relational database built upon a triple store.

The place of the binary relational database is established with reference to other database models, and a detailed description of a new triple store implementation is presented, together with a definition of the architecture.

The use of a model, which reflects the performance of the triple store database, is described, and the results of performance investigations are presented. In the first, the use of more than one sort order in the triple store database is analyzed, and the use of two sort orders is found to be optimal. In the second, the effect of compression in the triple store is considered, and compared with other approaches to compressing the non-index portion of a database management system.

In conclusion, the model successfully predicts the effect of using two sort orders, and this was confirmed upon subsequent incorporation into the database. It is also found that significant performance gains can be made by the use of compression in the triple store. It is shown that by extending the compression algorithm even greater gains could be made. In addition, it is found that by keeping the design of the database as simple and pure as possible, a foundation for a variety of higher level views can be achieved, leading to the possibility of the triple store being used as the foundation for new databases.

# Contents

**List of Figures** v

**Acknowledgements** vi

**1 Introduction** 1

**2 Background: Databases and Compression** 4

**2.1 Database Comparison** 4

2.1.1 Hierarchical Databases 5

2.1.2 Network Databases 6

2.1.3 Relational Databases 8

2.1.4 Object-Oriented Databases 11

2.1.5 Object-Relational Databases 14

2.1.6 XML Databases 15

2.1.7 Limitations and an Alternative 16

2.1.8 Binary Relational Databases 18

**2.2 Compression in Databases** 21

**3 The Triple Store and Binary Relational Databases** 24

**3.1 A Binary Relational Database** 24

3.1.1 Sets and Domains 24

3.1.2 Entities and Attributes 25

3.1.3 Relations and Terminology 27

3.1.4 An Example Database 29

3.1.5 The Sets of Relations, Mappings, Formats and Sets 31

**3.2 The Triple Store and the Lexical Store** 32

3.2.1 Identifiers 32

3.2.2 Lexical Store (or Semantic Store) 33

3.2.3 The Triple Store 35

3.2.4 How the Triple Store and the Lexical Store Work Together 35

**3.3 Comments on the Triple Store** **37**
3.3.1 The Need for Sorting 37
3.3.2 Sorting and Indexing the Triple Store 37
3.3.3 Compression 38
**3.4 Comments on the Lexical Store** **38**
3.4.1 Allocation of IDs 38
3.4.2 Sorting and Indexing the Lexical Store 39
3.4.3 Small or Large Sets 39
**3.5 Using the Metadata** **40**
3.5.1 Building the Query 40
3.5.2 Executing the Query 41
**3.6 Summary** **41**

**4 Implementation** **42**
**4.1 Assumptions and Scope** **42**
**4.2 Introductory Definitions** **42**
**4.3 Rules of the Architecture** **44**
4.3.1 Formal Definitions 45
4.3.2 Rules 46
4.3.3 Observations and Consequences of Rules 47
4.3.4 Further Objectives 47
**4.4 Formats** **47**
4.4.1 Identifiers 47
4.4.2 Full Identifiers versus Shortened Identifiers 48
4.4.3 Lexicals 48
4.4.4 Triples 49
4.4.5 Special Values (System ID Constants) 49
**4.5 Interfaces** **50**
4.5.1 The Programming Interface 50
4.5.2 End User Interfaces 51
4.5.2.1 General purpose interface 51
4.5.2.2 Interface to support recruitment agency application 52
4.5.2.3 Data explorer web interface 54
4.5.2.4 SQL interface 55

**4.6 Preliminary Performance Comparison** **56**
**4.7 Other Aspects** **56**
4.7.1 Locking and Robust Cursors 56
4.7.2 Caching and Storing to Disk 57
**4.8 Discussion of Alternative Approaches** **58**
4.8.1 Identifiers 58
4.8.2 Relations 59
4.8.3 Sets are Disjoint 60
4.8.4 Mappings 60

**5 Optimizing Data Storage for Performance** **61**
**5.1 The Model** **62**
5.1.1 Summary of the Model 62
5.1.2 Extent and Limitations of the Model 62
5.1.3 Key Aspects of the DBMS being modelled 63
5.1.3.1 The cache 63
5.1.3.2 Data store sizes 63
5.1.3.3 Data retrieval 65
5.1.3.4 Formulae for triple store 65
5.1.4 Calibration and Validation 66
5.1.4.1 Block retrieval times 66
5.1.4.2 Processing times 69
5.1.4.3 Validation 69
**5.2 Investigating Sort Orders** **70**
5.2.1 Results 70
5.2.2 Discussion 71
**5.3 Investigating the Effect of Compression** **71**
5.3.1 Compression in Databases 71
5.3.2 Towards a Compression Algorithm 72
5.3.2.1 The scope for compression in the triple store 73
5.3.2.2 Possible approaches 74
5.3.2.3 The block mask algorithm 74
5.3.2.4 Evaluation of algorithm 75

**5.4 Modelling the Performance Improvement due to Compression 76**

5.4.1 The Database 76

5.4.2 Establishing the Model 77

**5.5 Results of Compression Investigation 78**

5.5.1 Direct Access 78

5.5.2 Sequential Access 79

5.5.3 Discussion of Compression Results 80

**6 Conclusions and Discussion 81**

**6.1 Effect of Compression on Performance 81**

6.1.1 Further Compression in the Triple Store 82

**6.2 Performance Modelling with a Spreadsheet 83**

**6.3 The Triple Store – Achievements and Further Work 85**

6.3.1 Demonstrating the Advantages of the Triple Store Implementation 85

6.3.2 The Triple Store Database and Object Orientation 86

6.3.3 Concurrency Control in the Triple Store 86

6.3.3.1 Predicate locking 87

6.3.3.2 Predicate locking problems 88

6.3.3.3 Predicate locking in the triple store 88

**6.4 Could the Future be Binary Relational? 89**

**Appendix A 91**

**References 99**

# List of Figures


| | | |
|---|---|---|
| **Figure 2.1** | A meaningless relation | 18 |
| **Figure 2.2** | A binary representation | 19 |
| | | |
| **Figure 3.1** | Sets of entities | 25 |
| **Figure 3.2** | One entity set and three attribute sets | 26 |
| **Figure 3.3** | Cartesian product for relation (R) | 27 |
| **Figure 3.4** | Solution set (R*) for R | 27 |
| **Figure 3.5** | Table with four columns | 28 |
| **Figure 3.6** | Relations between entity and attribute sets | 29 |
| **Figure 3.7** | A personnel database | 30 |
| **Figure 3.8** | Metadata | 32 |
| **Figure 3.9** | Part of a lexical store | 34 |
| **Figure 3.10** | Part of a triple store | 36 |
| | | |
| **Figure 4.1** | Definitions | 45 |
| **Figure 4.2** | Rules | 46 |
| **Figure 4.3** | TD Operations | 50 |
| **Figure 4.4** | General Purpose Interface | 51 |
| **Figure 4.5** | ER diagram for recruitment agency | 52 |
| **Figure 4.6** | Data model for recruitment agency | 53 |
| **Figure 4.7** | Recruitment application | 54 |
| **Figure 4.8** | Web interface – Data Explorer | 55 |
| **Figure 4.9** | Status table record contents | 57 |
| | | |
| **Figure 5.1** | Variation of data rate with block size for sequential reads | 67 |
| **Figure 5.2** | Variation of block read time with block size for random reads | 68 |
| **Figure 5.3** | Variation of data rate with block size for random reads | 68 |
| **Figure 5.4** | Variation of random read time with file size | 69 |
| **Figure 5.5** | Effect of sort order on query times | 70 |
| **Figure 5.6** | Example of the block mask algorithm | 75 |
| **Figure 5.7** | Triple retrieval time with 256 kilobyte cache | 79 |
| **Figure 5.8** | Triple retrieval time with 1 megabyte cache | 79 |

## Acknowledgements

My most sincere thanks are due to Norman Winterbottom, my guide and mentor. Norman was a pioneer of binary relational databases, and has been the inspiration behind this work, and collaborator in many parts of it. Without his never-failing enthusiasm, patience, and encouragement, this thesis would never have come to fruition.

My warmest thanks are also due to Professor Anthony J G Hey, the original supervisor of this research, for his support, and to Dr Denis A Nicole, who became joint supervisor, and who has gave me great assistance in the final preparation of the thesis.

Finally, this work would never have been completed without the unfailing encouragement of my wife, Margaret, who kept on believing in the project, even when the going was tough. Thank you to you most of all.

# 1 Introduction

This thesis introduces a new approach to understanding the issues relating to the efficient implementation of a binary relational database built upon a triple store. A model has been built which reflects the performance of a new implementation of a triple store database. The model has been used to explore the potential benefits of extending the implementation. In turn, this has led to a new understanding of the benefits of compression within the triple store, which were discovered to be much greater than in a traditional relational database.

The idea of a binary relational database is not new. Research in the area has been ongoing, and various products have been brought to market, ranging from IBM's Data Mapping Program, shipped in the 1980s, to a database being shipped in 2002 by Lazy Software Ltd.

The advantages of basing a binary relational database on a triple store include the following aspects:

- The triple environment is essentially uniform, leading to efficiency and economy
- A considerable amount of processing can be carried out within the triple store itself, without manipulating a large number of data items
- The underlying model needs relatively simple code to access and maintain the data
- The uniformity of the triple store yields very significant compression opportunities
- The triple store also has the potential to be made completely self-tuning, which would be a significant benefit for both larger and smaller users.
- The uniform data structure is easier to spread onto multiple disks for parallelization

The background to the development of databases, and in particular the binary relational model, is introduced in Chapter 2, and previous work on compression is also reviewed. Chapter 3 gives a detailed description of the new triple store, while Chapter 4 covers the architecture behind it. Chapter 4 also describes a number of different interfaces that have been developed to provide a front end to the database.

The thesis then presents, in Chapter 5, the novel modelling technique used for performance investigation, and contains the results of two sets of experiments using this model, the first on the effect on performance of storing data in different ways, and the second on the effect of compression in the triple store. There has been much work on the compression of indexes in databases, but much less on the compression of the actual data in a database. The triple store mechanism provides a unique opportunity for high data compression, and the modelling exercise led to some new and interesting conclusions.

Chapter 6 then brings together the major aspects of the project, presenting conclusions concerning the effect of compression on performance, the benefits of taking the selected approach to performance modelling, and the achievement of this implementation of the triple store, before asking the question, "Could the future be binary relational?"

* * * * * * * * *

This work was carried out in collaboration with Norman Winterbottom, a Visiting Research Fellow at the University of Southampton. The design was developed jointly, and I then produced all of the documentation, some of which forms part of this thesis. Chapter 4 captures the essence of the design decisions that were made, and Appendix A gives a flavour of the detail. Coding was shared; Norman Winterbottom carried out the greater part of this task, but I wrote the code to handle the cache and the interface to the disk. I put considerable effort into many iterations of system and performance testing, which became a discovery process in itself as performance characteristics became clear and the design was refined. This led to the development and calibration of the performance model, which was entirely my work, and I undertook all of the subsequent investigations with the model.

An important part of the project was also to demonstrate that the triple store could form the basis for a variety of external views. I involved some final year students in this aspect, and I was responsible for guiding them through their work to successful conclusions. This provided further insight and feedback as to the facilities which were required at the programming interface.

I have written two papers with Norman Winterbottom reporting new results as an outcome of this work. The first [OCon00], which deals with sort orders in the triple store, has already been published. The second [OCon02] has been submitted to SIGMOD Record for consideration. This deals with the effect of compressing non-index data in the triple store database, and the results contrast significantly with other recent published work. The work reported in the two papers is presented in Chapter 5 of this thesis.

# 2 Background: Databases and Compression

For the last twenty years, relational databases have commanded much attention in terms of commercial investment (for example Oracle, DB2, MS SQL Server) and academic interest, followed at some distance by object-oriented databases. However, now that the limitations of the relational approach are beginning to restrict end users who wish to handle many more varieties of data, some object-oriented databases are being marketed (for example ObjectStore, Cache, Objectivity/DB), and object-relational databases, which aim to combine elements from both traditions, are being brought to market by the traditional RDBMS vendors. The increasingly widespread use of XML is also driving traditional vendors to offer various levels of support, and new 'native-XML' databases such as Tamino are appearing. At the same time, there is an increasing demand for databases to run on parallel platforms, to manage ever-growing volumes of data, to handle high transaction rates for on-line transaction processing, and to enable new applications such as on-line analytical processing.

In Section 2.1, the current technologies are reviewed. Standard texts such as [Elm00], [Dat00] or [Gray93] provide full descriptions, but the particular interest here is to contrast the way in which different database models handle the various kinds of relationship between data items, how the data is actually stored, and to introduce some of the issues that must be addressed. The binary relational approach is then introduced and contrasted, to provide the context and background for the work that has been undertaken

In Section 2.2, previous work on compression in databases is reviewed.

## 2.1 Database Comparison

### 2.1.1 Hierarchical Databases

The first hierarchical DBMS was IBM's IMS, with its own data language, DL/1. There are no references that precede the shipment of the product, but [McG77] gives an overview of IMS and some aspects were later formalized [Bjo82]. IMS is fully described in a large collection of IBM manuals. Later work considered the incorporation of

hierarchical structures within a relational database [Gys89, Jag89]. Another major commercial offering was System 2000, now marketed by SAS inc.

A hierarchy is a tree structure, containing a number of nodes or records (also called segments). As data is added to a hierarchical database, trees of records containing related data values are created. Each of these is called a 'hierarchical occurrence'. Each hierarchical occurrence has one root record, and contains all of the child records that relate to this particular root record.

There are various ways of holding a hierarchical occurrence in storage and on disk. One common way is to use a 'hierarchical record', which stores the data in a 'hierarchical sequence'. For each record within the sequence (except the root) there may be a varying number of instances, so it is necessary to store the type indicator with each record, so that the data can subsequently be interpreted.

The hierarchical sequence has the effect of storing many of the data items that are closely related close together in storage, so that retrieval is then efficient. However, some requests for data will entail gathering data that is spread into different parts of the hierarchical record, or in other records, and large amounts of data must then be retrieved.

Relationships between different elements of the data may be held in three ways:

- within the same record
- via a Parent-Child relationship
- via a Virtual Parent-Child relationship

Within the database, these are implemented by virtue of being in the same hierarchical record or by physical pointers from record to record. They may also be combined in various ways. However, the use of both the hierarchical record and physical pointers means that the logical data structure is carried over into the physical representation to some extent. Subsequent changes to the database schema may therefore involve reorganizing all of the data in the database.

Hierarchical databases permit the reduction or elimination of duplicate data, and while some real world systems are naturally hierarchical, extensions such as virtual pointers allow most other systems to be modelled. However, many real world systems do not fit the model easily, and m:n relationship types can be represented only by adding redundant records or by using virtual parent-child relationships and pointer records. There are various other restrictions, and database schemata can get very complicated. Some of the first major commercial systems were built using hierarchical databases. Many major companies have made a big investment in systems using hierarchical databases, and these will still be in use for many years.

### 2.1.2 Network Databases

*(Note: The word 'network' here refers to the organization of the data. It has nothing to do with whether the database is distributed over a communications network or not.)*

The Network Model was defined by the DBTG (Data Base Task Group) of the CODASYL (Conference on Data Systems Languages) committee from 1971 onwards, and is often referred to as the CODASYL network model [CODASYL], or as the DBTG model. The various aspects were defined in considerable detail, to form a standard upon which vendors could base their implementations. However this followed earlier work by Charles Bachman and others during the development of the first commercial DBMS, which was the Integrated Data Store (IDS). Bachman also introduced his 'Bachman diagrams' for describing relationships in a database [Bac69].

As in the hierarchical model, data is stored in records, which are classified into **record types.** However, the network model allows more complex data items to be defined. In addition to records containing simple and composite single-valued attributes, the model also permits records with simple multivalued attributes, which are known as **vectors**, and records with composite multivalued attributes, which are known as **repeating groups**.

The network model also supports **virtual** (or **derived**) **data items**. The value of a derived data item is not stored in the database, but is calculated 'on the fly' from other data that is stored in the database, according to a procedure supplied by the user.

In the network model, the construct called a **set type** is used to represent *relationships*. Sets in the Network model are *not the same* as mathematical sets.

A set type is a description of a 1:N relationship between two record types. Each set has

- A **set name**
- One **owner record** from the owner record type
- A number (zero or more) of related **member records**. In addition, the member records are ordered. (The order is immaterial in a mathematical set.)

In the relational model, described in the next section, a table is a normal mathematical set of tuples, all of which are of the same type and which represent instances of an *entity*, together with its attributes. In the network model, each set represents *one* instance of a *relationship*, and the set type represents a relationship type. For example, there might be a set type for the relationship 'MANAGES', in which each set would contain one manager, and all of the employees who work for him. This is a fundamental difference of approach.

In general, records will not be stored as contiguous sets in the database. Indeed, if records participate in more than one set, it is plainly impossible to store them in this way, and for performance reasons, alternatives may be preferable anyway. Typically, records are linked into sets using some sort of pointer structure.

A set instance is often kept as a **ring (circular linked list)** linking the owner record and all of the member records. The records carry an internal identifier to indicate which is the owner and which are the member records. Each record also has one pointer field to point to the next record in the ring for *each* set of which it is a member.
Other representations of sets include the following:

- Doubly linked circular list:- pointers go forwards and backwards
- Owner pointer representation, in combination with a ring:- each record has an additional pointer pointing to the owner
- Contiguous member records
- Pointer arrays:- owner has an array of pointers to the members. Usually used with owner pointer representation
- Indexed representation:- a small index is kept with the owner for *each* set occurrence

The relationships between different elements of the data may be held in three different ways (or in combinations):

- within the same record
- via set membership
- via linked records belonging to more than one set

With network databases duplicate data can be reduced, or eliminated. It is easier to model many systems with networks than with hierarchies, and the relationships are *explicitly* modelled, but the actual storage of such data models is more complicated and may be less space efficient than with hierarchical data models. Record-at-a-time processing means the programmer has to do more work than with set-oriented processing, and navigation is carried out in program logic, which the programmer must implement. This also means that it may not be possible to alter the structure of the database without changing programs, so that data independence is not fully supported.

Major commercial systems have been built using network databases, and as with hierarchical databases, user enterprises have made a big investment in systems using network databases.

### 2.1.3 Relational Databases

The relational model of data was introduced by Codd [Codd70], who went on to introduce relational algebra and develop the theory of relational databases in a series of papers [Codd71, Codd72, Codd72a, Codd74]. There has been much research on various aspects

of the relational model. The Peterlee Relational Test Vehicle (PRTV) [Todd76] was an experimental database that directly implemented the relational algebra operations. The PRTV was developed at IBM's research centre which existed for a while at Peterlee in County Durham. However, the dominant way to access relational databases soon became the use of Structured Query Language (SQL), originally known as SEQUEL [Cha76].

In the relational world, a database structure is presented to the user as a collection of relations or tables, with each table organized into rows and columns. It is important to realize that tables are the logical structure in a relational system, not the physical structure. The DBMS is free to use any or all of the usual structures underneath the covers.

In a relational database, the same information can be structured in many ways by assembling the data into different tables. The number of tables can range from one to almost the total number of columns in the database. The database designer has to decide how tables should be laid out. However, the original relational theory imposes the *first normal form constraint* which requires the attribute values in a relation to be only *atomic*. This means that a column value must not itself be a tuple or a set, and rules out repeating groups. (Post-Relational databases relax this restriction.) To avoid certain update anomalies it is necessary to go further and reduce tables to *second* and *third* normal forms, and possibly higher. Second normal form applies to relations with composite keys: for a relation to be in second normal form, it must be in first normal form and every non-primary-key attribute must be fully functionally dependant on the primary key, and not on just a part of the key. To be in third normal form, a relation must be in second normal form, and there must be no transitive dependencies between any non-primary-key attribute and the primary key.

The query language in relational database systems is *declarative* - the user states what he wants, and the DBMS works out how to get it. The various operations (such as joins, restrictions and so on) always produce another relation as their output, and this relation can be input to further relational operations if required. It is interesting to note, however, that once a join has been carried out, there is no longer any guarantee that the resulting relation is fully normalized - it frequently will not be. SQL is the *de facto* standard relational language, but there are other approaches and front-end technologies that allow user-friendly access to relational databases, often based on the Query By Example (QBE)

paradigm. With QBE, the query is formulated by filling in templates of tables displayed on a computer screen. QBE was one of the first graphical query languages for database systems. It was developed at IBM Research [Zlo75], and can be used, for example, with DB/2. It is also the approach used for one of the query interfaces into Microsoft Access and Paradox.

These approaches to data manipulation contrast strongly with the navigational approach needed to work with hierarchical and network databases. With the navigational approach, the programmer essentially follows the pointers inside the database, but there are no pointers inside a relational database. The DBMS must do the work of retrieving related items of data.

When it comes to storing the data, each row in a table typically becomes a *stored record*, a string of bytes with a prefix containing system control information and up to *n* stored fields, where *n* is the number of columns in the base table. Internally, each record has a unique *record id (RID)* within a database. The RID consists of the page number and the byte offset from the start of the page of a slot that, in turn, contains the record's starting position within the page. Thus records within a page can be reorganized without changing their RIDs. Each stored field includes three elements:

- A prefix field that contains the length of the data
- A null indicator prefix that indicates whether the field contains a null value
- An encoded data value

The only relationships that are represented physically in relational databases are between the items of data that are members of the same record (tuple). Beyond this, relationships are not represented physically within the database. They must be rediscovered / reconstructed when a query or update is needed, by combining tables on the basis of looking for equal values in specified columns of each table. In setting up the tables in the first place, it is therefore necessary to duplicate data in different tables in order for this to be possible. To ensure that data is kept consistent, integrity constraints are needed, in particular, referential integrity.

Thus, relationships between different elements of the data may be discovered in two different ways:

- by finding related items within the same record
- by combining ('joining') two or more relations to form a new relation

or by combinations of these.

The major advantage of relational databases, then, is that the database is *perceived by the user* as tables, and that access is declarative, not navigational. Access is therefore not dependent upon physically implemented pointers. To obtain reasonable performance, the database management software must have powerful capabilities to interpret the user's queries and optimize their execution. There is a far greater degree of data independence than with the previous two models, although there is significant data duplication.

It is perhaps important to realize, however that "the relational model of data was not really a model at all, but rather a theory" [Dar96], based on mathematical sets. Mathematical relations do not necessarily model the real world, and there are a number of restrictions. Design and normalization require significant skill, and DB management software is much more complex. Relational systems impose the first normal form constraint, which means that the object space must be mapped onto a collection of 'flat' relations (i.e. tables). With this approach much of the inherent semantics of complex object composition is lost, and one needs to perform foreign key joins to reconstruct a complex object.

Relational databases currently occupy by far the largest part of the marketplace, with large numbers of vendors supplying DBMSs and complementary products. The major offerings are now IBM's DB2 [DB2], Oracle [Oracle], Microsoft's SQL Server [SQLServ] and Sybase [Sybase], but there are many others, specializing in certain markets such as the desktop or in application areas like geographic information systems.

### 2.1.4 Object-Oriented Databases

The models discussed so far are quite successful for handling straightforward business data. However, there are other applications that have different requirements. These

include engineering design and manufacturing (CAD/CAM), image and graphics databases, scientific databases, geographic information systems, multimedia databases and so on. These applications require support for structures that are more complex, or for unstructured objects such as images, which need non-standard routines to handle them. Object-oriented databases (ODBMSs) have been developed to handle these.

There is no single way of implementing ODBMSs, and at this point in time, researchers and manufacturers have developed a wide range of different approaches [Kho93]. The Object Data Management Group [ODMG] is now providing a focal point for some degree of convergence and the development of standards [Catt95]. However, one characteristic is that ODBMSs normally support a persistent programming paradigm. The programmer treats objects in the same way, regardless of whether they are stored in the database or not. Objects are *persistent* if they are stored permanently in the database or *transient* if they only exist during execution of a program.

In the case of ObjectStore [ObDes], for example, the system is closely integrated with the C++ language, and provides persistent storage facilities for C++ objects. This choice was made to avoid the *impedance mismatch* problem between a database system and its programming language, where the structures provided by the database system are distinct from those provided by the programming language. Objectivity/DB [ObDb] is a distributed ODBMS. It is designed for mission-critical and production environments, and claims to offer high performance, virtually unlimited scalability, and interoperability across all major platforms and operating systems.

One of the fundamental concepts of object orientation is *Object Identity*. Object identity organizes the *objects* or instances of an application in arbitrary graph-structured object spaces. Identity is the property of an object that distinguishes the object from all other objects in the application. In a complete object-oriented system each object is given an identity that will be permanently associated whatever structural or state changes take place. Identity is independent of location, or address. Object identity provides the most natural modelling primitive to allow the same object to be a sub-object of multiple parent objects.

With object identity, objects can contain or refer to other objects. Object identity clarifies, enhances and extends the notions of pointers, foreign keys, and file names. Using object identity, programmers can dynamically construct arbitrary graph-structured composite or complex objects - objects that are constructed from sub-objects. Objects can be created and disposed of at run-time. If using an ODBMS, objects can become persistent and be reaccessed in subsequent programs.

Two types of reference semantics exist between a complex object and its components at each level:

- **Ownership semantics** applies when the sub objects of a complex object are encapsulated within the complex object and are hence considered part of the complex object.
- **Reference semantics** applies when the components of the complex object are themselves independent objects but at times may be considered part of the complex object.

Storage mechanisms vary widely within ODBMSs. A typical implementation might store items of data linked by ownership close together on the disk, in a similar fashion to the data in a 'record'. Items linked by reference would, in C++ terms, be represented by a pointer. Internally, the ODBMS is likely to make use of object identifiers to resolve such pointers, so that there is no dependence on any underlying physical structure.

In summary, ODBMSs support complex objects and extensible data types, with complex relationships between objects. The use of object identifiers divorces the 'labelling' of entities from the data values associated with the entities, and makes for a much cleaner approach.

ODBMSs are not yet as sophisticated as RDBMSs, however, and tend to be tightly linked to a single language, most often C++. For performance reasons, ODBMSs normally run in the same address space as the applications, whereas RDBMSs require an address space switch, which provides a major security benefit. Although commercial users are increasing steadily, this is likely to stay a niche market, and many of the best features are now being adopted by object-relational databases.

### 2.1.5 Object-Relational Databases

Relational DBMSs provide excellent support for simple data and simple to somewhat complex queries. Object-oriented DBMSs provide efficient support for certain classes of applications on complex data, but without many relational 'goodies' like queryablility, security, database administration and so on. Neither the current RDBMSs nor the current ODBMSs find it easy to meet the growing demands of new applications requiring complex querying on complex data, including multimedia data. Object-relational systems (ORDBMSs) aim to combine the benefits of the RDBMSs with the modelling capabilities of the ODBMSs, thus providing support for complex queries on complex data [Ston96].

There has been a concerted standards effort to extend SQL-92 to provide the extra facilities needed to support ORDBMSs, resulting in SQL3 [App B in Dat00]. SQL3 aims to be a computationally complete language for the definition and management of persistent, complex objects. It includes generalization and specialization hierarchies, multiple inheritance, user-defined data types, triggers and assertions, support for knowledge-based systems, recursive query expressions, and additional data administration tools. It also includes the specification of abstract data types, object identifiers, methods, inheritance, polymorphism, encapsulation, and all of the other facilities normally associated with object data management.

The production of ORDBMSs is mainly driven by the RDBMS vendors, by adding function to their existing products to offer some of the above facilities, often starting with the support of data types such as Binary Large Objects (BLOBs) and Character Large Objects (CLOBs). BLOBs include images, video clips and sound tracks that have an internal structure that cannot be handled by any of the traditional database approaches. CLOBs are documents containing text, probably formatted in some way, maybe using HTML or XML (see below). With demand for complex data and complex queries in traditional business applications, ORDBMSs seem to be a natural progression for RDBMSs, but it remains to be seen to what extent they are accepted in the marketplace.

It is also worth noting that various research efforts, such as Gamma [DeW90], Volcano [Gra90] (which formed the basis of the parallel implementation of Informix) and others

have found ways to adapt traditional relational database engines to run on parallel hardware, and object extensions are being added on top of these foundations.

### 2.1.6 XML Databases

Extended Markup Language [XML] is increasingly being used to provide a flexible way to capture the structure of documents and their contents, and enable information transfer between users across networks. The relational database format is well suited for stable information structures, and data that fits well into fully populated rows and columns, but XML documents do not conform to this paradigm. Relational database vendors tend to offer XML support as an add-on, and still do most of the work with SQL and tables. The information is stored in tables, and only converted to XML when needed.

Native XML databases use XML as the primary means for structuring, organizing and storing information. Like SQL, XML provides full searching and indexing, but XML goes one step further by letting users modify the structure of a document without destroying any data already stored in it. There is then no need to perform XML-RDBMS translations or transformations.

A number of native XML databases are now appearing on the market. One is Tamino [Tam]. (Tamino is an acronym for Transactional Architecture for Managing Internet Objects. Tamino is also the hero in Mozart's *The Magic Flute*!) An additional capability here is that it can scan a well-formed XML document and work out what the structure is, whereas relational data must always have its structure already specified before being loaded into a database. Another native XML database is XIS (eXtensible Information Server) from eXcelon [Exc].

While the native XML database (XDBMS) is an attractive idea, existing organizations with large amounts of data already stored in RDBMSs are unlikely to convert their databases in the short term, but XML will be increasingly used to convey information from one database to another.

### 2.1.7 Limitations and an alternative

Relational databases demand that data be partly decomposed into a series of flat tables, which are typically stored record by record on to the disk. Elaborate normalization rules have to be followed to ensure that this partial decomposition is carried out correctly, and it is not difficult to lose information in the process. Object-oriented databases work in terms of complex objects, which have to be flattened before storage on to disk, and this can easily result in non-uniformity in data access, as some related data items are bound to end up widely separated on the disk, while others will be held in the same block.

An alternative approach in either case is to fully decompose the data [Kho96]. At first sight, this may not appear to be promising from a performance point of view, but studies [Cop85, Kho87] have shown that it can work very well, and commercial databases based on this philosophy have been successfully marketed. One of these is the so-called 'Data Mapping Program' from IBM [DMP82] (a fully functional database management system), and recently, Lazy Software has brought to market a database based on what is termed "The Associative Model of Data" [Lazy], which is essentially an extension of the binary relational model. Sybase have also brought out a search tool which extracts data into a binary relational format.

Decomposed data may be stored in 'two-column tables', as in MONET [Bon96], [Bon99], but a more radical approach, which was used in the Data Mapping Program is to build a Triple Store to hold the data. A key aspect of this is the separation of relationships from the data. As a result, much of the internal query processing can be performed on uniform identifiers, rather than somewhat heterogeneous data strings, leading to simplification of coding within the database, and potential performance advantages.

Above a fully decomposed data store, it is possible to build object-oriented or normal (n-ary) relational databases or, as in the case of the Data Mapping Program, a binary-relational view can be offered to end users. The binary-relational model is described in [Fro86]. It offers a very easy-to-use and intuitive approach for end users. Another characteristic of such a database built on a triple store is that catalog (or data dictionary) information is automatically contained within the store [Shar78], and administering this is

very straightforward. In an n-ary relational database, relationships can only be reconstructed if duplicate data is held in more than one table, and often in several tables. If a data item changes its value, then several tables need to be updated. If a binary relational database is based on a triple store, data values can be held in a separate 'lexical' store (see chapter 3). The triple store itself only holds tokens representing data items, and these tokens will be duplicated as necessary to build relationships. The value of a data item will be stored in one place only and not duplicated. The number of triples in the triple store is related to the number of instances of each field in the database, and the structure means that there is indexed access to every field in the database.

Initial work has shown that a Triple Store also forms a very natural basis for holding data in the object-oriented environment, in other words, to build an object-oriented database. By separating the relationships from the data, a more uniform pattern of access is obtainable. It appears that the Triple Store could be an engine that would support a number of different models at the user-interface level.

One further development that has taken place since the original work on a triple store database is the advent of widely deployed and relatively affordable parallel computers. The triple store is a very promising architecture for parallelization, due to its simplicity and uniformity. No application structure is apparent in the triple store, in contrast with n-ary relational databases. In the latter case, careful consideration has to be given to the partitioning of tables and the collocating of the various portions to reflect application activity, while attempting to strike a balance between various requirements. A triple store can be split in the optimum way to facilitate the internal processing needs of the DBMS.

The original implementation of the Triple Store database [DMP82] was eclipsed by the arrival of the n-ary relational databases, leaving many aspects of the approach completely unexplored. Now that the limitations of the n-ary approach are becoming more apparent, it is time to re-open examination of this simple and elegant model, to see what new ideas and insight can be gained.

### 2.1.8 Binary Relational Databases

The binary relational approach has been introduced above, and will be fully explored in Chapter 3. Interest in binary relations goes back to the earliest days of databases. In [Senko77], a paper which gives a fascinating insight into the debates in progress at the time, there is discussion about the way data should be presented at the logical level. The Data Independent Accessing Model (DIAM) [Senko73] was a data model which included a logical-level as one of its levels, and gave rise to much subsequent development. DIAM was developed further in DIAM II [Senko80]. [Senko77] presents two contrasting views of data. Figure 2.1 shows what he calls a 'meaningless relation', originally discussed in [Sch75]. The question posed in this example is what the appearance of FRIEND and SALARY in the same relation implies. Does SALARY imply the "salary of the FRIEND" or the "salary of the EMPLOYEE"? Such a relation is without semantic meaning, and something must be added to make the meaning clear to the user, perhaps in terms of constraints.

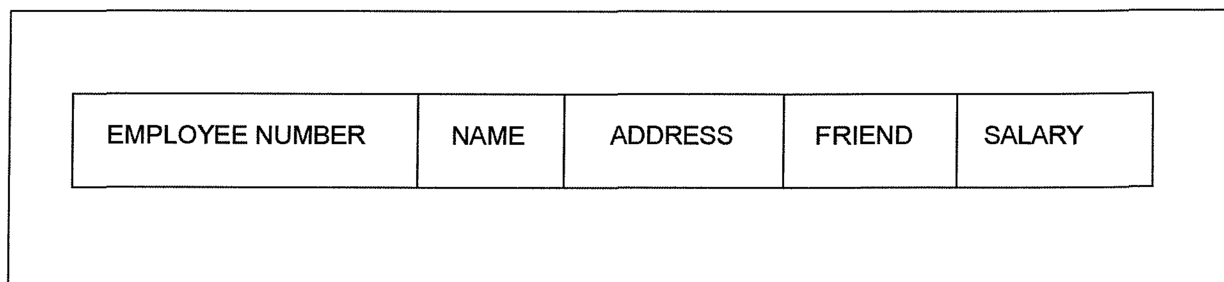



**Figure 2.1** A meaningless relation

Senko contrasts this with the binary representation shown in Figure 2.2. In this case, it is clear that SALARY is a direct attribute of EMPLOYEE, and it is only indirectly related to FRIEND by way of EMPLOYEE, illustrating that binary relations seem to be a fitting representation of facts.

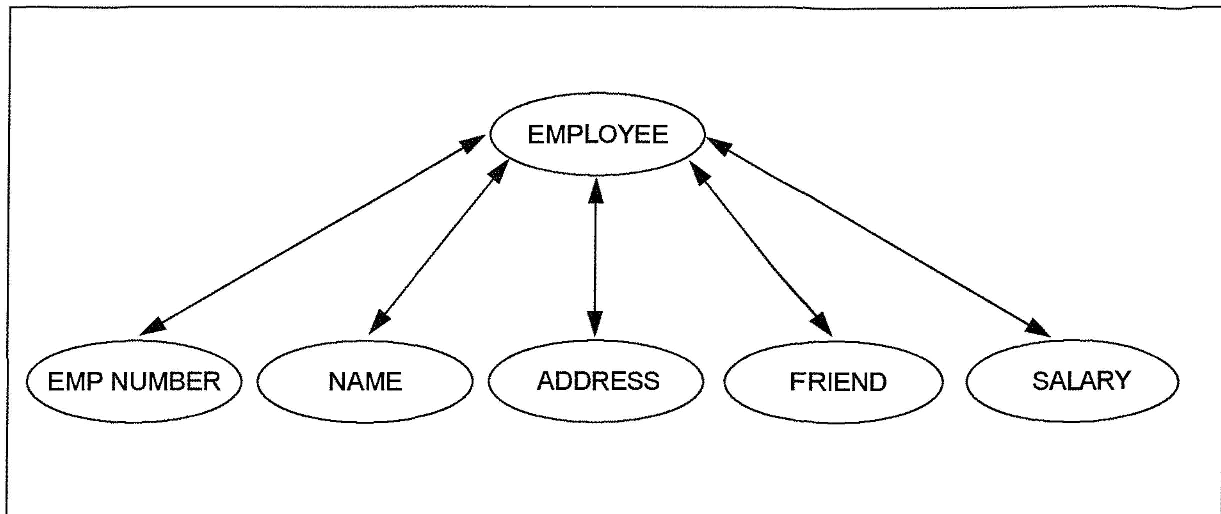



**Figure 2.2** A binary representation

In 1976, Chen had proposed the entity-relationship (ER) model for data modelling [Chen76] and database design, which adopts a very similar approach. Today, the ER model is widely used. The popularity of the ER approach stems from the fact it captures the entities that are being modelled, together with their attributes and the relationships between them, in a manner that is easy to understand and in line with human intuition, while providing a formalism from which the designer can then move forward. To move from an ER design to a relational schema, a set of rules must be carefully followed. Entities will be represented by tables, but while some relationships can be captured using fields duplicated between these tables, others will have to be represented by additional tables. Whereas it is a somewhat complex process to move from an ER model to arrive at a relational schema, it is a very simple step to move from ER to a binary relational database design because the two approaches are so close. (There is an example of this process in Section 4.5.2.2.)

If a binary relational view is attractive at the logical level, the question then arises as to how to implement such a database at the physical level. The hierarchical and network views of data carry the logical structure right down to the physical level. Adopting this approach for binary relations leads to large numbers of two column tables, and the associated processing would be prohibitively costly. Early implementations such as [Lev67], [Ash68] and [Feld69] followed this line of attack by storing each set of binary relations in a separate file. Titman [Tit74] took a different approach, by storing triples in ordered arrays, and the Non-programmer Database (NDB) [Shar79] was directly

influenced by this work. (The contents of the triples are not necessarily the same in all implementations. The triples used in the current work differ from these earlier databases.)

In 1982, a paper was published by Frost [Fro82] which reviewed several research efforts under way at the time. Frost begins his introduction to binary relational storage structures as follows: "Any part of the universe, no matter how complex, can be thought of as a set of binary relationships. Consequently, a structure, within which representations of such relationships can be manipulated, is logically sufficient as the storage mechanism for a general purpose database system." He also remarks that at that time "the binary relational view of the universe is increasingly being used during the database analysis stage of database design."

Frost describes a number of different structures for holding the triples, including:

- holding triples in ordered arrays, with one array for each relation
- holding the triples as one set, replicated three times and held in three separate hash tables, keyed on different combinations of two out of the three items
- a linked list structure
- a master file of triples, with a set of inverted lists. For each entity, there are three inverted lists giving the addresses of the entity either as subject, relation, or object.

Subsequent work has been based on various approaches. The work by Copeland and others referred to earlier [Cop85, Kho87] used binary relations held in what is referred to as a decomposed storage model (DSM). Their performance comparisons with an n-ary storage model (NSM) showed that similar results could be achieved, with each model having particular strengths and weaknesses. There were also projects based on the use of special hardware, such as the FACT Machine [McG80]. In [Shar88], the Universal Triple Machine (UTM) was introduced, in which the data repository consisted of two stores: the name store and the triple store. In [Mar92a], the implementation of an object-oriented database (Oggetto) layered over a triple store is described, which is capable of handling the four tasks for an object-oriented database outlined in [Atk87], and the same author reports the development of a 3D graphical interface in [Mar92b].

Currently, there are at least two groups actively working with binary relational models. In Amsterdam, a novel database server known as Monet has been developed. [Bon96] gives an overview of Monet, and discusses how it is being used to support ODBMS applications. Monet is also based on a decomposed storage model (DSM), which is implemented using 'binary association tables' (BATs). These are very much the same as the first approach described by Frost. A BAT is a two-column table representing one binary relation, and the database will have multiple BATs. Monet is designed to perform all operations in main/virtual memory. For databases which exceed available physical memory, Monet relies on virtual memory, by memory mapping large files. Work has also been carried out with Monet on parallel machines, and a prototype has been run on an IBM SP machine.

At Birkbeck College in London, the Triple Store Architecture Research (TriStarp) Project [King90], [TriStarp] was set up to explore the use of the binary relational data model at all levels in a database system. The triple store in this case was built based on three-dimensional Grid Files, in which each dimension represents one of the three elements of a triple. Recent work has concentrated on the higher levels of the DBMS such as Fudal [Sut95], a functional database language, and GQL [Pap95], a Graphical Query Language, and other aspects rather than on the underlying structure.

As mentioned above, one of the major binary relational databases described by Frost was the Non-Programmer Database Facility (NDB) [Shar79], which was subsequently marketed as "DMP" [DMP82]. The work on NDB led to further research [Giles82], [Fitz90], which supported and validated the approach taken. The present research carries forward the idea of the triple store, but with a different structure, and explores new areas that have not been dealt with before, in particular, the aspect of data compression.

## 2.2 Compression in Databases

The potential benefits of compressing data in a database are twofold. First, there is the obvious outcome of saving space on the disk or other storage medium. However, with storage becoming ever cheaper, this is no longer so important. The second benefit is to achieve an improvement in performance, by reducing the number of disk accesses. This implies a trade-off between the reduction in disk I/O and the cost of compressing and

decompressing the data. Decompression is particularly important in this context; data is only compressed once, but decompression is likely to be required time after time for query processing as well as for the delivery of the final answer.

Many efforts in the context of relational databases have dealt with compression in the index. The benefit of compressing indexes in a database has long been established, as described, for example, in [Wag73] (VSAM) and [Com79] (B-Trees). In an index, successive entries are sequenced, and various techniques such as prefix compression and suffix compression have been employed, as described in standard works such as [Gray93] or [Ram00]. Appropriately chosen strategies can reduce the size of the index, and as long as the cost of processing compressed index entries can be contained, faster retrievals can be achieved.

When the data itself is considered, the picture is not so clear cut, as the structure that exists in indexes does not generally apply. A major decision is the level at which to compress. It is possible to compress at the block level, the tuple level, or at the level of individual fields. The potential cost of having to decompress a whole block or tuple can outweigh any benefit. [Gold98] shows that if the UNIX 'gzip' facility (based on the Lempel-Ziv algorithm [Ziv77]) is used to compress a page, it will take longer to 'gunzip' it than to read the page from disk. Previous work, such as [Gra91, Ray95 and Gold98], has shown that compression in databases needs to be very fast, and also needs to be fine-grained. This leads to consideration of compression at the field level.

This is the approach taken in [Wes00], where fields are compressed into a specially formatted tuple, using a 'light-weight' approach, where only some of the fields are compressed. Integers and dates are compressed using null suppression and encoding of the resulting length of the compressed integer [Roth93]. For long strings, the authors consider the use of Huffman coding [Huff52], Arithmetic coding [Wit87], or the LZW algorithm [Welch84]. If order preservation is needed, then techniques such as those proposed in [Blas76, Ant96] are suggested. Westmann et al [Wes00] describe how the storage manager, the query execution engine and the query optimizer may be extended to deal with the compressed data, in the context of a TPC-D benchmark database [TPC95]. Their results show significant speed-up for long-running TPC-D decision support queries,

but they remark that they do not expect to see any benefit for short On-Line Transaction Processing (OLTP) queries.

Chen et al [Chen01] point out that many fields in the typical relation in fact contain short text strings, which are not compressed effectively by the algorithms listed above. They have devised a Hierarchical Dictionary Encoding (HDE) strategy that intelligently selects the most effective compression method for string-valued attributes. Chen et al then apply this to the problem of compression-aware query optimization, and demonstrate speed-up using a TPC-H benchmark database [TPC99], which again involves long-running decision support queries.

Both of these recent approaches deal with queries in a 'traditional' n-ary relational database with large numbers of records. These queries require heavy processing in query optimization and execution. This results from the fact that in a relational database, relationship information has to be re-discovered from the data stored in the relations every time a query is executed. If the data items are compressed, they will generally have to be decompressed to allow query processing to proceed, although it is sometimes possible to work with attribute values in their compressed form.

In this thesis, the focus is on the issues which come to light when a binary relational database architecture is employed, in this case, built on a triple store. Here, information about relationships is stored separately from the data items, so that query processing can be carried out without the need to decompress data items along the way. Only the data items finally presented in the answer need to be decompressed at the end of a query. Query execution does, however, require extensive processing of triple store records, and the question is then whether compression in the triple store can benefit this processing.

A new compression algorithm has been developed for the database. Using this, records could be compressed when initially inserted into the triple store, but from then on, processing would be carried out efficiently without needing to decompress the records again. A modelling exercise, described in Chapter 5, was carried out to explore the extent of the performance improvement, with interesting results.

# 3 The Triple Store and Binary Relational Databases

This chapter introduces in an informal manner the concepts on which this project is based. Since the triple store lends itself very naturally to supporting binary relational databases, the implementation is based on this model.

## 3.1 A Binary Relational Database

### 3.1.1 Sets and Domains

Real world objects can be categorized in **sets**, such as the set of all of the staff in an organization, the set of all products manufactured by a company, or the set of children belonging to one person. Any member of a set has certain properties that help to describe it. For example, a person may be described by weight, height, age and so on. There are also other properties that tell us something about set members, such as who their manager is or parents are, or which flavour of ice cream they prefer. Traditionally, these properties are termed 'attributes'. In a standard (n-ary) relational database, one item, for example a person, will be identified by some unique key, e.g. staff number.

Considering the example of 'salaries', it is clear that the set of all salaries can be divided into many different, possibly overlapping sets, such as 'managers' salaries', 'women's salaries', or 'salaries of full time staff'. However, if calculations involving salaries are carried out, it is clear that common rules apply. The format of the data must always have exactly two places after the decimal point, and while a salary can be multiplied by a number such as 1.05 to calculate a 5% increase, it makes no sense to multiply two salaries by each other. One might also specify that all salaries must be divisible by 12, to make it easy to compute monthly payments. It is useful, therefore, to develop rules for dealing with salaries. Data to which common rules can be applied is said to belong to the same '**domain**'. Prices would not belong to the same domain as salaries, as, although some rules are in common, such as the format of the data, other rules such as 'divisible by 12' would not apply. All items of data in one domain have common attributes, and common rules governing their processing.

### 3.1.2 Entities and Attributes

It is possible to think of attributes as 'adjectives' that describe an item. Age or sex can easily be thought of in these terms. However, this does not really work for all attributes. Is a person's manager an adjective? Surely not - a manager is also an item in his/her own right, and may even be a member of the same domain 'employee' as the member of staff in his/her department.

An alternative approach is to describe all characteristics as 'entities'. An entity might be 'staff number', 'name', 'height', 'telephone number', 'manager', 'skill' or 'eye colour'. Each of these entities is contained within its own set. There will therefore be a set containing 'staff numbers', and to describe a person, we will need to have links or connections to the appropriate member of each other set, such as 'name' or 'age' (Figure 3.1).

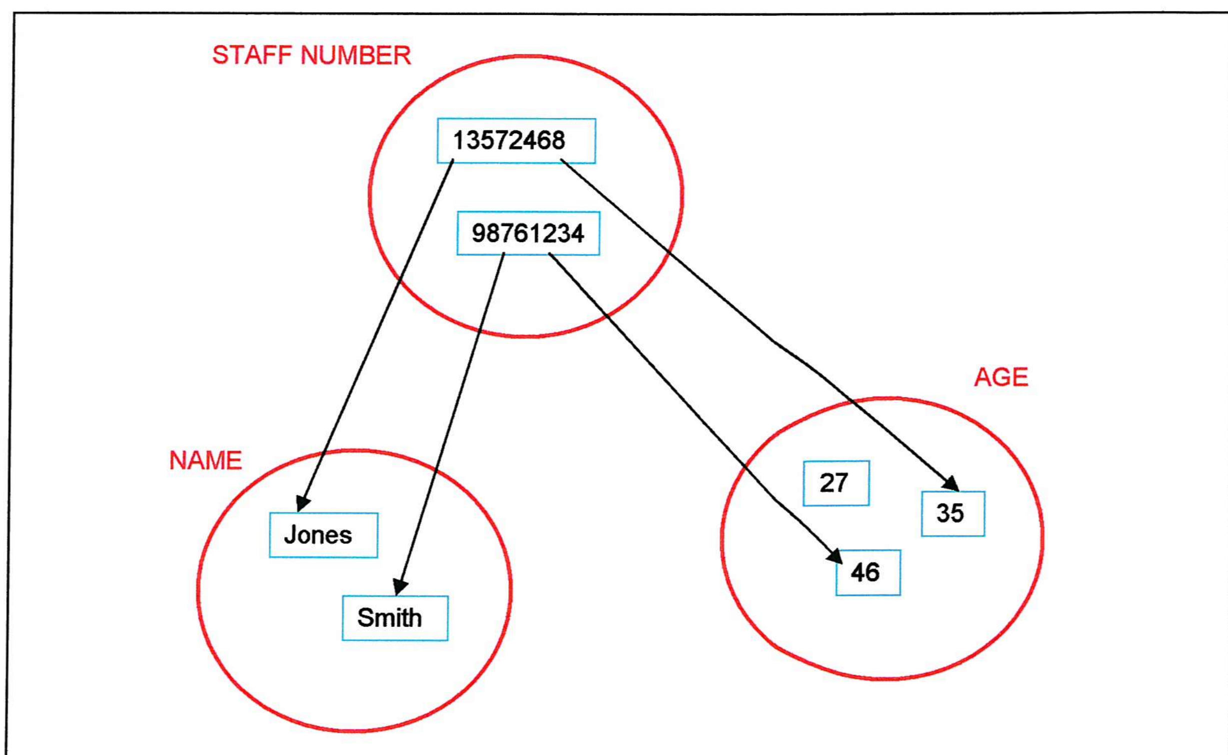



**Figure 3.1** Sets of entities

In thinking about entities, it is important to be clear in the definition and meaning. When talking about telephones, for example, do we mean 'the number in the book' or 'the gadget on the desk'? When people move offices, they may take the number and/or gadget

or neither with them. The domain of descriptors needs to be specified. In the case of telephone numbers, this might be 'exactly 4 digits, beginning with 5 or 6'.

In the earlier implementation of a binary relational database known as NDB, the above approach of dealing with **all** characteristics as entities was taken. This places no constraints on the data, and worked successfully. However, there is some value in placing constraints on the way that data is being used, if this reflects the situation in the real world better. A person can have blue eyes, and a height of six feet, but 'blue' cannot have a height. Therefore, it has been decided to distinguish attributes and entities for this implementation, as follows:-

> An **entity** relates to some 'thing' in the real world being modelled. The entity does not have any properties, except for an identifier ('ID') internal to the database, until attributes have been 'attached' to it.
>
> An **attribute** describes some property of an entity. An attribute will belong to a domain (e.g. 'colours'), and will have some value (e.g. 'blue'). The attribute is completely defined once its domain and value are known. An attribute cannot be attached to any other attribute, but only to an entity (Figure 3.2).

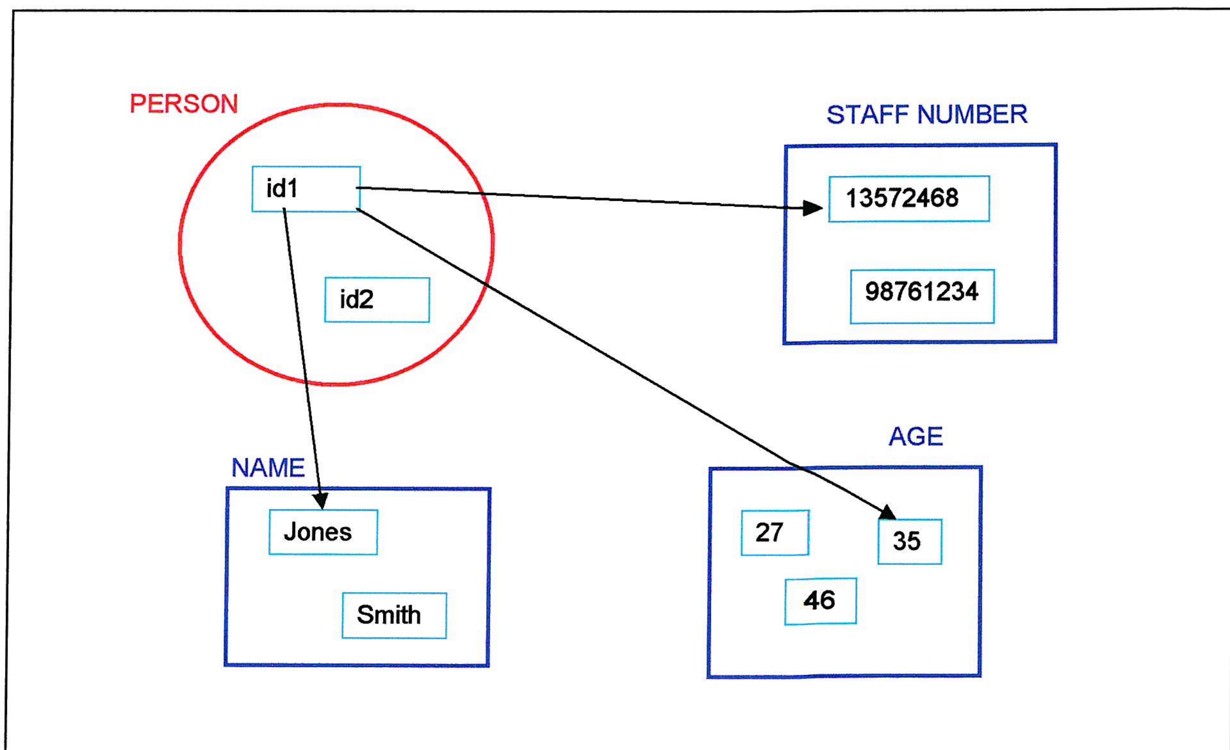



**Figure 3.2** One entity set and three attribute sets

### 3.1.3 Relations and Terminology

Set Theory includes the concept of *relations*. To illustrate this, consider two sets, A which is the set of playwrights, and B which is the set of plays. We could define a propositional function, P(x, y) = "x wrote y" which would be either true or false for any combination of the elements (a, b) of the two sets. For example,

P(Shakespeare, Hamlet) = "Shakespeare wrote Hamlet" is true, while
P(Shakespeare, Faust) = "Shakespeare wrote Faust" is not true

A *relation R* consists of

1) a set A
2) a set B
3) P(x,y) in which P(a,b) is either true or false for any ordered pair (a, b)

R is called a *relation from A to B*. Relations are not limited to just two sets, but can include any number. The *solution set R** of the relation R consists of the elements (a, b) in the Cartesian product, A X B (Figure 3.3) for which P(a, b) is true (Figure 3.4).

| | |
|---|---|
| Shakespeare | Hamlet |
| Shakespeare | Faust |
| Goethe | Hamlet |
| Goethe | Faust |

**Figure 3.3** Cartesian product for relation (R)

| | |
|---|---|
| Shakespeare | Hamlet |
| Goethe | Faust |

**Figure 3.4** Solution set (R*) for R

In the case of the triple store, which will be introduced shortly, each line only ever relates members from each of two sets, so that all relations are binary, and the implementation is termed **'binary relational'**. However whilst the low level implementation is binary relational, the view of data presented to the end user could be very different.

Consider now a conventional relational database 'relation' or table which has four columns: S#, SNAME, STATUS, CITY (Figure 3.5). This relates values belonging to 4 sets. The *database* relation (table) is the *solution set* of the much bigger relation that contains all *possible* combinations of the members of the 4 sets (i.e. the Cartesian

product). The *solution set* includes only the rows which contain *valid* combinations of the values.

| S# | SNAME | STATUS | CITY | |
|---|---|---|---|---|
| 1234 | Jones | OK | London | Valid |
| 3456 | Williams | Owes Us | Cardiff | Valid |
| 8976 | McIntosh | OK | Glasgow | Valid |
| 8675 | Smith | Insolvent | Birmingham | Valid |
| | | | | |
| *1234* | *Jones* | *OK* | *Brighton* | *Invalid* |
| *8976* | *Williams* | *Insolvent* | *Cardiff* | *Invalid* |
| ... | ... | ... | ... | *Invalid* |

**Figure 3.5** Table with four columns. The database would only contain the solution set, i.e. the 'valid' rows

It is usual for conventional relational databases to store individual tables separately. Using a triple store, however, it is possible to store all relational information in one single 'table'. An individual line in the triple store contains one **relationship** or **connection** (which is one occurrence of the appropriate propositional function with the value 'true'). All of the (non-metadata) lines in the triple store that have the same value in the relation column correspond to one conventional database relation or table.

For ease of reference, from this point on, the term **connection** will normally be used to refer to an individual instance of a relationship, and the term **relation** to refer to the collection of all of the connections of the same type, which is in line with the standard usage of the word in database literature. Each relation has a name, such as 'HasManager', or 'SkillName', and each connection uses this name to show what sort of connection it is (Figure 3.6).

NOTE: In standard relational database tables, each row has several ('n') columns and expresses the relationship between the 'n' attributes described in the columns. Hence they are often referred to as '**n-ary relational'** databases.

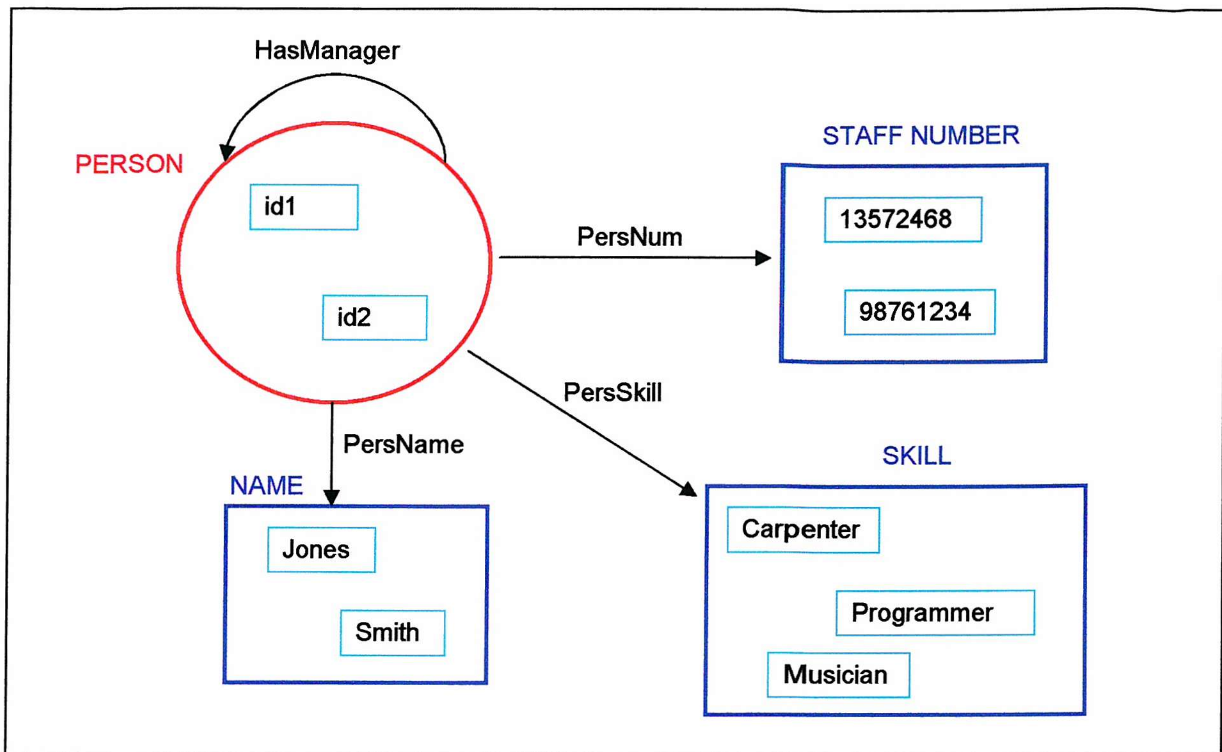



**Figure 3.6** Relations between entity and attribute sets

### 3.1.4 An Example Database

To describe a real-world item completely, it is necessary to establish the connections from any starting point, for example, a person in the set of 'person' entities, to the appropriate member of any other relevant entity or attribute set.

Although the connections are binary, there may be multiple connections between an entity in one set and entities in another set. A person might have several different skills, and a telephone may be shared by many users. There are four possible mappings that may be used to describe this, which are:

| | | | |
|---|---|---|---|
| 1:1 | 'one to one' | m:1 | 'many to one' |
| 1:m | 'one to many' | m:n | 'many to many' |

For example, 1:m mapping means that one member of one set may have several connections, each of them formed with a different member of another set. However, it is not necessary to use all four of these mappings. If connections can be traversed in either direction, then only one of the mappings 1:m or m:1 is needed. In addition, an m:n

mapping can always be replaced by introducing an additional entity and using two m:1 mappings. In the diagram that follows, all mappings are either m:1 or 1:1.

In certain situations, it may be required that for every member of a set, there *must* be a connection to another set - a *mandatory* 1:1 mapping. It might be useful to extend the above list to include such a mapping, but this has not been done at the moment. The direction of the connection needs to be expressed in some way, as shown by the arrows on the diagram. Any implementation will need to adopt a convention for this, and also provide support for connections between members of the same set, as demonstrated by the 'HasManager' connection. The direction of the connection makes clear who is the manager, and whom is being managed. However, it will be possible to use a connection in either direction to traverse the database, so that if we wish to discover which employees report to a particular manager, we can use the 'HasManager' connection in the reverse direction.

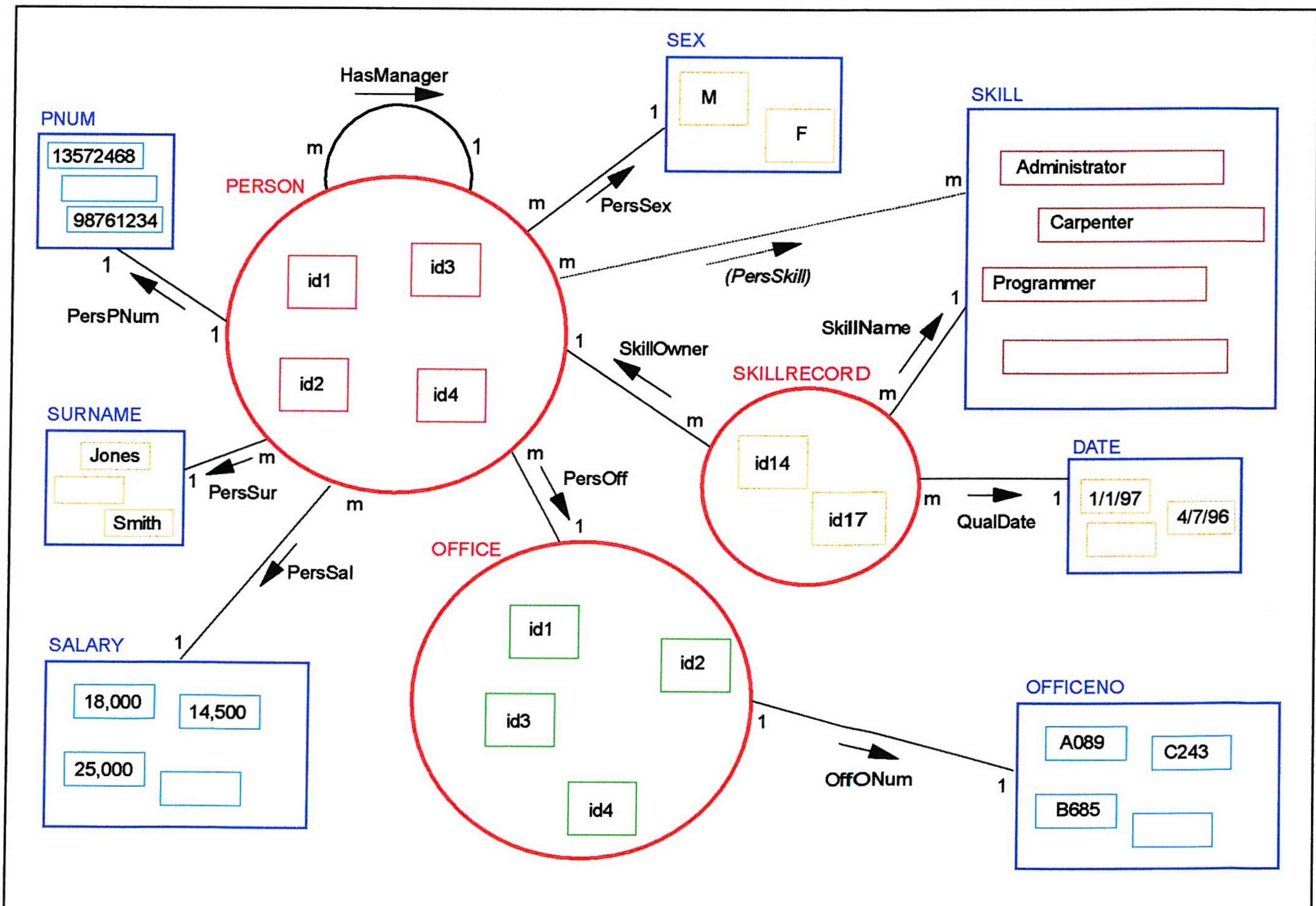



**Figure 3.7** A personnel database

All of the above points are illustrated in Figure 3.7, which shows a simple database. There are ten sets in the example.

- Three Entity Sets: Person, Office, SkillRecord
- Seven Attribute Sets: Pnum, Surname, Salary, Sex, Skill, Date, OfficeNo

There are various connections between the sets, which are listed below. Note that all of the connections are given either 1:1 or m:1 mappings. The m:n mapping that would have been required by the connection 'PersSkill' has been eliminated by introducing the SkillRecord entity. This has the benefit that attributes that relate to this Person to Skill connection (termed 'intersection data'), such as the date that a person qualified with a new skill, can now be added to the database.

| Relation name | Description |
|---|---|
| • PersPNum | Person's personnel number - 1:1 |
| • PersSur | Person's surname - m:1 (people may have the same name) |
| • PersSal | Person's salary - m:1 |
| • PersSex | Person's sex - m:1 |
| • HasManager | Person's manager - m:1 |
| • SkillOwner | The person to whom a SkillRecord relates - m:1 |
| • SkillName | The name of the skill - m:1 |
| • QualDate | The date on which the person acquired this skill - m:1 |
| • PersOff | Person to office - m:1 (some people share offices) |
| • OffONum | Office to office number - 1:1 |

This approach, which we might call a 'data map', provides a very simple way to analyse and think about relationships between data.

### 3.1.5 The Sets of Relations, Mappings, Formats and Sets

The relations also form a set, as do the mappings, formats and the sets themselves (i.e. there is a set of sets). These may also be represented within the model using exactly the same approach, as shown in Figure 3.8.

- Each set has a name and a format for the data items in the set
- Each member of the 'relation' set has a name and a mapping
- Each member of the 'relation' set describes connections from a member of one set to a member of another set - 'from' and 'to' indicating the direction of the connections

So data about the data, usually termed '**metadata**', can be stored in the same database as the data itself.

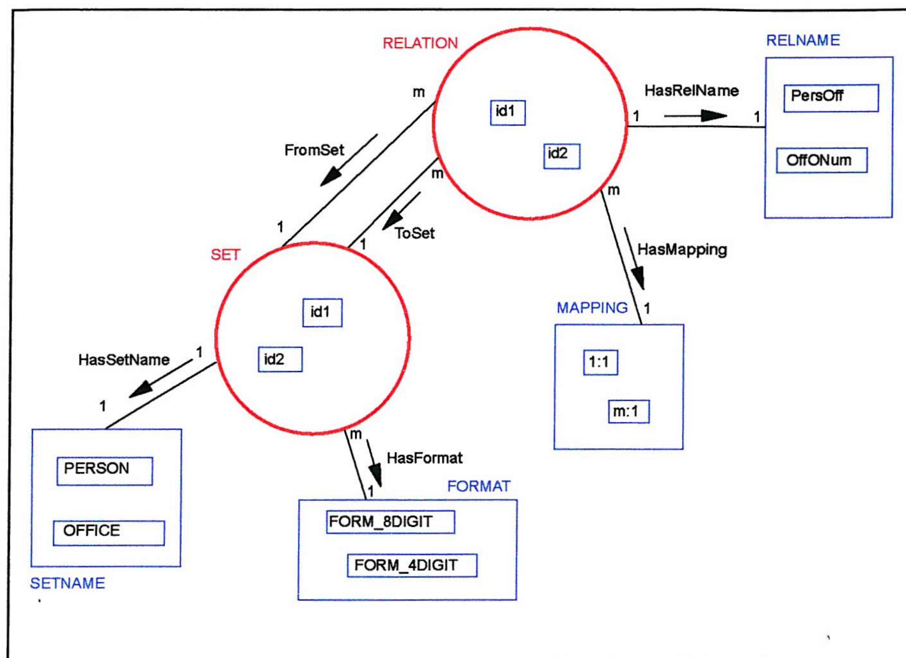



**Figure 3.8** Metadata

## *3.2 The Triple Store and the Lexical Store*

### 3.2.1 Identifiers

In the preceding example, entities were given identifiers (IDs), but attributes were shown by their values, and connections by their relation names. To preserve symmetry in the triple store, and to permit performance to be enhanced in the implementation, attributes and relations are also given identifiers. A second table, which is termed the 'lexical store', is then used to translate the attribute IDs to and from actual values when needed.

In the following example, identifiers of the form

**set-name | id-number**

are used to demonstrate the principle. The form of identifier used in the implementation is defined in Chapter 4.

### 3.2.2 Lexical Store (or Semantic Store)

The lexical store provides the 'bridge' from the triple store to the outside world. The lexical store maps the internal identifiers to the values that they represent.

Entities do not have values, and therefore do not appear in the lexical store. Entities include the sets in the database. For reference, the example is using internal identifiers for sets as follows. In operation, the end user has no knowledge of any internal identifiers.

| | **Set** | **Internal Identifier** |
|---|---|---|
| • | PERSON | SET_id1 |
| • | OFFICE | SET_id2 |
| • | SKILLRECORD | SET_id3 |
| • | PNUM | SET_id11 |
| • | SURNAME | SET_id12 |
| • | SALARY | SET_id13 |
| • | SEX | SET_id14 |
| • | SKILL | SET_id15 |
| • | OFFICENO | SET_id16 |
| • | DATE | SET_id17 |

Figure 3.9 shows part of the lexical store for the database in Figure 3.7, including the entries needed for the metadata shown in Figure 3.8.

| Identifiers | Values |
|---|---|
| | |
| PNUM_id1 | 13572468 |
| PNUM_id2 | 98761234 |
| PNUM_id3 | 92847557 |
| PNUM_id4 | 87364512 |
| | |
| SURNAME_id1 | Smith |
| SURNAME_id2 | Williams |
| SURNAME_id4 | Jones |
| | |
| SEX_id1 | M |
| SEX_id2 | F |
| | |
| SALARY_id7 | 25,000 |
| SALARY_id9 | 18,000 |
| | |
| SKILL_id31 | Carpenter |
| SKILL_id32 | Administrator |
| SKILL_id34 | Programmer |
| SKILL_id35 | Xylophonist |
| | |
| DATE_id74 | 01/01/1997 |
| DATE_id47 | 04/07/1996 |
| | |
| OFFICENO_id2 | A089 |
| OFFICENO_id44 | B685 |
| | |
| RELNAME_id1 | PersPnum |
| RELNAME_id2 | PersSur |
| RELNAME_id3 | PersSal |
| RELNAME_id4 | PersSex |
| RELNAME_id6 | HasManager |
| RELNAME_id7 | PersOff |
| RELNAME_id8 | OffONum |
| RELNAME_id10 | SkillOwner |
| RELNAME_id11 | SkillName |
| RELNAME_id12 | Qualdate |
| RELNAME_id20 | HasSetName |
| RELNAME_id21 | HasFormat |
| RELNAME_id30 | HasRelName |
| RELNAME_id31 | HasMapping |
| RELNAME_id40 | FromSet |
| RELNAME_id41 | ToSet |
| | |
| SETNAME_id1 | PERSON |
| SETNAME_id2 | OFFICE |
| SETNAME_id3 | PNUM |
| SETNAME_id4 | SURNAME |
| SETNAME_id8 | OFFICENO |
| | |
| FORMAT_id8 | 8DIGIT |
| FORMAT_id1 | 1BIT |
| | |
| MAPPING_id2 | m:1 |

**Figure 3.9** Part of a lexical store

### 3.2.3 The Triple Store

The triple store is a table which is designed to contain the connections. Each line in the table is a 'triple' containing

- The ID of the relation to which the connection belongs (the 'relId')
- The ID of the item (entity) that the connection leads 'from' (the 'fromId')
- The ID of the item (entity or attribute) that the connection leads 'to' (the 'toId')

Figure 3.10 shows part of the triple store for the database in Figure 3.7, including the entries needed for the metadata shown in Figure 3.8.

There are two sorts of entry in the triple store:

- Entries describing the connections shown Figure 3.7, which for clarity are shown in the upper part of Figure 3.10

- Entries describing the data itself shown in Figure 3.8 - the metadata - which are shown in the lower part of Figure 3.10

  Metadata in a database is often termed 'system catalog' data. It is also sometimes called the data dictionary, although this term is also used to refer to a separate repository of information about the data in an organization.

### 3.2.4 How the Triple Store and Lexical Store work together

To find Smith's office number:

1. Go to Lexical Store and find "Smith"
   - returns SURNAME_id1
2. Go to Triple Store with SURNAME_id1 and REL_id2 (PersSur)
   - returns PERSON_id2
3. Go to Triple Store with PERSON_id2 and REL_id7 (PersOff)
   - returns OFFICE_id3
4. Go to Triple Store with OFFICE_id3 and REL_id8 (OffONum)
   - returns OFFICENO_id44
5. Go to Lexical Store with OFFICENO_id44
   - returns the Office number - B685

| Relation | From | To |
|---|---|---|
| | | |
| REL_id1 | PERSON_id1 | PNUM_id2 |
| REL_id2 | PERSON_id1 | SURNAME_id4 *(Jones)* |
| REL_id3 | PERSON_id1 | SALARY__id9 |
| REL_id4 | PERSON_id1 | SEX_id1 |
| REL_id6 | PERSON_id1 | PERSON_id3 |
| REL_id7 | PERSON_id1 | OFFICE_id3 |
| REL_id10 | SKILLRECORD_id14 | PERSON_id1 |
| REL_id11 | SKILLRECORD_id14 | SKILL_id34 |
| REL_id12 | SKILLRECORD_id14 | DATE_id74 |
| REL_id10 | SKILLRECORD_id17 | PERSON_id1 |
| REL_id11 | SKILLRECORD_id17 | SKILL_id35 |
| REL_id12 | SKILLRECORD_id17 | DATE_id47 |
| | | |
| REL_id1 | PERSON_id2 | PNUM_id3 |
| REL_id2 | PERSON_id2 | SURNAME_id1 *(Smith)* |
| REL_id3 | PERSON_id2 | SALARY__id9 |
| REL_id4 | PERSON_id2 | SEX_id2 |
| REL_id6 | PERSON_id2 | PERSON_id3 |
| REL_id7 | PERSON_id2 | OFFICE_id3 |
| | | |
| REL_id1 | PERSON_id3 | PNUM_id4 |
| REL_id2 | PERSON_id3 | SURNAME_id2 *(Williams)* |
| REL_id3 | PERSON_id3 | SALARY__id7 |
| REL_id4 | PERSON_id3 | SEX_id1 |
| REL_id5 | PERSON_id3 | SKILL_id35 |
| REL_id7 | PERSON_id3 | OFFICE_id7 |
| REL_id10 | SKILLRECORD_id24 | PERSON_id3 |
| REL_id11 | SKILLRECORD_id24 | SKILL_id32 |
| REL_id12 | SKILLRECORD_id24 | DATE_id74 |
| | | |
| REL_id8 | OFFICE_id3 | OFFICENO_id44 |
| REL_id8 | OFFICE_id7 | OFFICENO_id2 |
| | | |
| | | |
| | | |
| REL_id20 | SET_id11 | SETNAME_id3 *(PNUM)* |
| REL_id21 | SET_id11 | FORMAT_id8 *(8DIGIT)* |
| | | |
| REL_id20 | SET_id2 | SETNAME_id2 *(OFFICE)* |
| | | |
| REL_id20 | SET_id16 | SETNAME_id8 *(OFFICENO)* |
| REL_id21 | SET_id16 | FORMAT_id4 *(4DIGIT)* |
| | | |
| REL_id30 | REL_id7 | RELNAME_id7 *(PersOff)* |
| REL_id40 | REL_id7 | SET_id1 *(PERSON)* |
| REL_id41 | REL_id7 | SET_id2 *(OFFICE)* |
| REL_id31 | REL_id7 | MAPPING_id2 *(m:1)* |
| | | |
| REL_id30 | REL_id6 | RELNAME_id6 *(HasManager)* |
| REL_id40 | REL_id6 | SET_id1 *(PERSON)* |
| REL_id41 | REL_id6 | SET_id1 *(PERSON)* |
| REL_id31 | REL_id6 | MAPPING_id2 *(m:1)* |

**Figure 3.10** Part of a triple store

## *3.3 Comments on the Triple Store*

### 3.3.1 The Need for Sorting

One of the over-riding considerations in any database system is to minimize the number of disk accesses. When searching for related data, response times are going to be much faster if all of the required data can be read in from disk at once. If the user wants to know all about PERSON_id1, it will be better if this information is not scattered randomly throughout the triple store. Best performance will be achieved if logically related items are close together physically, which can be aided by sorting both the triple store and the lexical store.

### 3.3.2 Sorting and Indexing the Triple Store

The three columns of the triple store can potentially be sorted in 6 different ways. Each of these has the effect of grouping related items together. For example:

- If the sort is based on the order: second column, first column, third column, in the above table, all of the connections concerning PERSON_id1 will be stored together. When data is read from a disk, a whole block is read at a time, which will contain many rows of the triple store. So when any of the rows relating to PERSON_id1 is retrieved into memory for processing, all of the connections concerning the person will almost certainly be brought in too, and further disk access is not needed to traverse the related data.
- If, however, the sorting is done on: third column, first column, second column, then all records for each Office_id would be grouped, so that people occupying the same office could be traced quickly.

To optimize processing, therefore, *several* sort orders (but not necessarily all 6) may be maintained within the triple store, using as many copies of each entry as there are sort orders. There is a trade-off to be made between performance and disk space, but if good compression techniques are used, properly sorted data will compress very significantly, so that the space overhead is not as large as might at first appear. One aspect of the project has been to determine how many sort orders it is worth maintaining.

In addition to sorting the triple store, there is also a need for indexes (e.g. a B-Tree or other index) to permit reaching the appropriate part of the triple store fast.

### 3.3.3 Compression

To take full advantage of the proposed structure, consideration must be given to opportunities for compression, and the techniques that could be applied.

If the triple store is sorted on the second column, all of the entries beginning PERSON_ will be together, and so on. It would therefore be necessary to store 'PERSON' only once with a count of the number of PERSON_ entries that follow. The same argument applies to all columns, hence the need to consider sorting in more than one way to get the maximum benefit. It is probably even possible to spot recurring groups of entries and compress these. The use of identifiers with two parts (set_id | item_id - see Chapter 4) lends itself well to this sort of compression, helping to conserve disk space.

## *3.4 Comments on the Lexical Store*

### 3.4.1 Allocation of IDs

A mechanism is needed for the allocation of IDs as attribute values are added to the lexical store. A number of algorithms are possible for this:- allocate numbers sequentially, allocate numbers randomly etc., as long as uniqueness within a set is maintained. Another possibility is to make the ID in the triple store equal to the literal value of the data.

A further strategy, which has been considered, is to allocate IDs in an order which reflects the natural sort order for the domain, where this exists (e.g., alphabetic or numeric). This would have the effect that when the triple store is sorted by ID, entries will automatically be sorted in an appropriate order for other processing. Also, if entries are placed in the triple store in the order of the data, then a *range* can be examined by locating the first and last values via the lexical store, indexing to the appropriate entries in the triple store, and

then working entirely within the triple store, knowing that all entries between the two limits satisfied the criteria.

However, the difficulty of allocating IDs as more values are added, without causing an entire renumbering operation to occur, which in turn would require every line in the triple store that referred to the attribute to be updated too, outweighs the possible benefit. It is not clear, either, that it is really desirable for the triple store to have the 'partial understanding' of the attribute values implied by the above approach. The triple store should really be completely indifferent to attribute values.

In the present implementation, IDs are allocated randomly within a range. The range is initially set small, so that IDs are fairly 'close' to each other to facilitate subsequent compression. If a range becomes too tightly filled, however, the range is dynamically expanded, and further expansion will occur as necessary.

### 3.4.2 Sorting and Indexing the Lexical Store

The lexical store needs to be sorted on the first column to bring all items of the same type together. This means sorting by the two parts of the identifier, SetName and ID. It may also be desirable to sort the lexical store on the second column - this was to be determined.

Indexing is needed to provide rapid access into the lexical store, and to support range and other queries. (This is no worse than for a conventional relational database, where secondary indexes are needed for searching and range queries on any field except the primary key.)

In particular, indexing into the second column is crucial. Indexes into this column have to cater for data stored in a variety of data formats. For example, surname, office_no, sex, and so on all have different formats.

### 3.4.3 Small or Large Sets

Some sets have a limited number of values, all of which could be preloaded into the lexical store in sorted order. Examples might be SEX, EYE_COLOUR and so on. These

may be thought of as 'closed' sets, and it is possible that a special algorithm might be used to allocate IDs for these.

Other sets have a potentially large number of values. Although they may still be technically 'closed', the upper limit of the possible values might be 'all of the rational numbers less than 10 million', or 'all possible combinations of 20 alphanumeric characters'. As far as the database is concerned, these sets are essentially 'open'. For these sets, decisions will have to be made about strategies for keeping them sorted within the lexical store, such as sorting every night, leaving spare space in the store, using a hashing technique and so on.

## *3.5 Using the Metadata*

### 3.5.1 Building the query

Consider again the example above - to find Smith's Office number:

To find Smith's Office number:

1. Go to Lexical Store and find "Smith"
   - returns SURNAME_id1
2. Go to Triple Store with SURNAME_id1 and REL_id2 (PersSur)
   - returns PERSON_id2
3. Go to Triple Store with PERSON_id2 and REL_id7 (PersOff)
   - returns OFFICE_id3
4. Go to Triple Store with OFFICE_id3 and REL_id8 (OffONum)
   - returns OFFICENO_id44
5. Go to Lexical Store with OFFICENO_id44
   - returns the Office number - B685

The user first has to instruct the database how to build the query. In a conventional RDBMS, this is accomplished by writing an SQL statement, or by using some sort of QBE interface. The user needs to understand the tables in the database, and the relationships between them in order to carry this out, and SQL statements can become extremely complex and hard to understand, except for the expert.

With a binary relational database, the user will have a data map, as described earlier, and will be provided with a friendly interactive front end with which to build up a query path. In general, it is much easier for the end user to achieve this than when using SQL. Once a particular query has been built, it should be possible to save it for re-use in the future. A programming interface will also be provided.

### 3.5.2 Executing the query

When the query path has been determined, the system can execute it using the metadata. There will be the following steps.

1. Find SURNAME_id

    User has supplied a string "Smith"
    Use metadata to check
        Is there a Set called SURNAME
        Is data-type etc valid ( SURNAME HasFormat)
        May want to do Domain check e.g. for enumerated domains
    Then go to Lexical Store to find SURNAME_id1 for "Smith"

2 Traverse the database from SURNAME_id1 to OFFICENO_id

*If this is an m:1 relationship (likely) then there may be more than one PERSON_id for SURNAME_id1.*

We will travel via the PERSON set in this example. The path is PersSur (inverse) 1:m, then PersOff m:1.

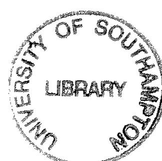
    Go to the Triple store to find all of the PERSON_ids for SURNAME_id1
    Go to the Triple Store to get the OFFICE_ids for the PERSON_ids
    Go to the Triple store to find the OFFICENO_ids for the OFFICE_ids

3 Retrieve the actual value of the Office number from the Lexical Store

    Go to the Lexical Store with the OFFICENO_ids to get the values.
    Go to the metadata again, using OFFICENO HasFormat to present the value to the user in the correct format.

## 3.6 Summary

This chapter has sought to introduce the concepts of the triple store, the lexical store and the binary relational approach. The next chapter will set out more formally the rules and architecture of the present implementation.



# 4 Implementation

This chapter begins with the working specification for the construction of the binary relational database based on a triple store, and then describes some of the end-user interfaces that have been constructed. The chapter concludes with discussion of other aspects concerning the implementation.

## 4.1 Assumptions and Scope

The implementation adheres as strictly as possible to the binary relational model, as described in the previous chapter.

The database is being built using object-oriented programming techniques in C++. (Note: this does not mean that the result will necessarily be an 'object-oriented database').

The approach concentrates on keeping the triple store in an optimum state to ensure rapid retrieval of information. This means that more work is needed when data is updated. In a real application, data is always read before being written, so the approach only impacts a maximum of 50% of the accesses adversely, and usually many fewer. A critical statistic will be the ratio of reads:writes for an application.

## 4.2 Introductory Definitions

The following terms are used:

1. The term *collection* to describe an arbitrary group of objects, not all drawn from the same domain.

2 The term *set* to describe a collection of items drawn from the same domain. (This is the normal usage in database discussions.) '*Set*' will normally be used to describe *all* of the members of a domain.

3. An *entity* relates to some 'thing' in the real world being modelled. The entity in the database exists for the purpose of providing a unique identifier for the real world thing. The entity itself does not have any properties, except for an identifier internal to the database, until attributes have been 'attached' to it. Entities may have relationships, called connections (see definition 6 below), one with another.

4. An *attribute* describes some property of an entity. An attribute will belong to a domain (e.g. 'colours'), and will have some *value* (e.g. 'blue'). The attribute instance is completely defined once its domain and value are known. An attribute value cannot be connected to any other attribute value, but only to an entity, i.e. there can be no connections between attributes, but only between attributes and entities (or entities and entities, as above).

5. An entity or an attribute value has an *identifier* ('*ID*'), which is assigned internally. The end user will never be aware of this *ID*. The *ID* is unique within a domain. If domains are unique, then specifying the entity or attribute by *domain_id.entity_id* is unique globally (within the database).

   The *ID* is *not* the value of an attribute. The value of the attribute is obtained from the Lexical Store. *IDs* are discussed further in Section 4.4 – Formats, below.

6. The relationships between entities or between entities and attributes are called *connections*. Connections only ever exist between two entities, or one entity and one attribute. All connections are therefore binary relationships.

7. A *relation* is the set of all connections of the same type, and a connection bears the name of the relation of which it is a member. (See Section 3.1.3 for further discussion.)

8. A *'triple'* consists of three full identifiers, one of which is a relation identifier, one is an entity identifier and the remaining one is either an entity identifier or an attribute value identifier. (See Section 3.2.3 for further discussion.)

9. *Triple store*: The triple store holds all of the triples in the database

10. A *'lexical'* is made up of two parts: a full identifier, and a data value. The format is discussed further below

11. *Lexical Store*: The lexical store holds all of the lexicals in the database

## 4.3 Rules of the Architecture

These are fundamental to the structure of the triple store and the lexical store. They apply equally to 'data' and 'metadata'.

### 4.3.1 Formal Definitions

| Definition number | Definition | Dependency on previous definitions | Notes |
|---|---|---|---|
| D1 | An **entity** is some 'thing' in the real world being modelled | | |
| D2 | An **entityId (entId)** is associated with each entity | D1 | 1 |
| | | | |
| D3 | An **attribute** is a property which can apply to an entity *(e.g. colour)* | D1 | |
| D4 | A **value** is a value that an attribute can assume *(e.g. blue)* | D3 | |
| D5 | A **lexicalId (lexId)** is associated with each lexical (attribute) value | D4 | |
| | | | |
| D6 | A **set** is a group of entities or a group of values | D1, D4 | |
| D7 | The **family** is the family of all of the sets in a database | D6 | |
| D8 | A **setId (setId)** is associated with each set | D6 | |
| D9 | An **entitySet** is a set of entities | D1, D6 | |
| D10 | A **lexicalSet** is a set of values relating to one attribute type | D4, D6 | |
| D11 | A **domain** is the set of all possible values that one attribute may take | D4, D6 | |
| | | | |
| D12 | A **connection** is a directed relationship with a given mapping from one entity to another, or from an entity to a value | D1, D4 | 2 |
| D13 | A **relation** is the set of all of the connections of a given type | D12 | 3 |
| D14 | A **relationId (relId)** identifies a relation (in other words, a type of connection) | D13 | |
| D15 | The **setOfRels** is the set of all of the relations in the database | D6, D14 | 4 |

**Figure 4.1** Definitions

*Notes on the Definitions:*

1. *An entity has no properties until attribute values are attached*
2. *This implies that all connections are binary*
3. *When a relation is created, the direction of the connections, the mapping, and the two sets being connected must be specified*
4. *A relation is itself an entity, which is described in the database through the metadata*

### 4.3.2 Rules

| Rule number | Rule | Dependency on definitions | Notes |
|---|---|---|---|
| R1 | SetIds are unique within the family | D7, D8 | |
| | | | |
| R2 | Each entity is a member of one and only one entitySet | D1, D9 | 1 |
| R3 | EntIds are unique within the entitySet of which the entity is a member | D2, D9 | |
| R4 | An entitySet contains only entityIds (no values) | D2, D9 | 2 |
| | | | |
| R5 | Each value is a member of one and only one lexicalSet | D4, D10 | |
| R6 | LexIds are unique within the lexicalSet of which the value is a member | D5, D10 | |
| R7 | A lexicalSet contains lexIds paired with (attribute) values | D5, D10 | |
| | | | |
| R8 | RelIds are unique within the SetOfRels | D14, D15 | |
| R9 | There is, at most, only one connection of a given type *from* any entity | D1, D12 | 3 |
| R10 | Connections between lexicalSets are forbidden | D10, D12 | |
| | | | |
| R11 | Domains are disjoint | D11 | |
| R12 | A domain contains data of only one type | D11 | |
| R13 | Values from one attribute domain are not comparable with values from another | D4, D11 | 4 |

**Figure 4.2** Rules

*Notes on the Rules*

1. *This implies that sets are disjoint. This is discussed further in Section 4.8.3*
2. *EntIds do not therefore appear in the lexical store, since there is no associated value*
3. *This means that all relationships are many to one (m:1), where m may also be 1 giving a one to one (1:1) relationship. However, it is possible to traverse relationships in the inverse direction. Many to many (m:n) relationships will not be supported. See Section 3.1.4 for further discussion*
4. *This means that a 'strong typing' environment will be enforced*

### 4.3.3 Observations and Consequences of the Rules

1. A relation must be defined before any triple that is based on it can be added to the triple store
2. There cannot be two identical lines in the triple store
3. There cannot be a lexId in the triple store which is not in the lexical store
4. There can be a lexId in the lexical store which is not in the triple store
5. There cannot be two (full) lexIds in the lexical store which are the same

### 4.3.4 Further Objectives

A query against a database should return a complete, self-contained database, including all of the relevant metadata.

This is analogous to a relational database, in which a query against a number of relations returns a relation, although in that case, the relation may not be normalized.

## *4.4 Formats*

### 4.4.1 Identifiers

A 'type' is defined for the identifier ('ID'), so that it can always be changed without impact to the rest of the code. Initially, identifiers map down to unsigned long integers (32 bits - 4 bytes). This gives a range of from 0 to 4,294,967,295 per set inclusive. Further types are then defined for entId, lexId, setId and so on, in terms of the basic ID. All IDs will therefore be based on the same underlying type, preserving symmetry throughout the database.

Certain values will be reserved for 'special' usage. These are IDs which are essential to the integrity of the database architecture.

### 4.4.2 Full Identifiers versus Shortened Identifiers

A full identifier (entId, lexId, relId, or setId) is made up of two parts, the ID representing the set of which the item is a member, and the ID of the item within the set. In the case of the setId, the first ID will be that of the setOfSets. In the current implementation, therefore, the full identifier will be 8 bytes long, made up of 2 4-byte IDs, although this would obviously change if the basic 'ID' type is changed. When new relations are defined, the first part of the full identifier will contain the ID of the setOfRelations; the second part, the newly assigned ID of the relation being added. In addition, entries must be made in the triple store to describe the fromSet, the toSet, and the mapping for this relation.

In principle, all items in the triple store and the lexical store would use full identifiers. However, as items will be held in sorted groups, especially in the triple store, considerable space can be saved by not repeating parts of the full identifier where not necessary. In the case of small sets (such as 'sets' and 'relations') this might mean that only one or two bytes need be stored for each entry.

In practice, the relId will always be the first of the three parts of the triple. The setId (which is the ID for the setOfRelations) is therefore not strictly needed. In addition, the relId implies the IDs of the two sets that are being connected, so that it is not necessary to store these either. Thus the triple need only contain the item identifier for all three parts. This is the approach that has been adopted, and each triple contains three 4-byte integers, which are the item identifiers for the relId, the fromId and the toId. (FromId and toId are described in Section 3.2.3.)

### 4.4.3 Lexicals

A *'lexical'* is made up of two parts: a full identifier and a data value.

Full identifiers, which will be lexIds, are allocated when items are added to the lexical store. The first part of the identifier will be the ID of the set (setId) to which the item is being added; the second part will be the newly allocated ID for this item.

The format of the data will be described via a hasFormat relationship, and the data will be stored in accordance with this, e.g. as a string, fixed length integer etc.

### 4.4.4 Triples

A *'triple'* consists of three full identifiers, one of which is a relation identifier, one is an entity identifier and the remaining one is either an entity identifier or a value identifier.

In addition, a fourth field will be added to indicate the sort order for a particular occurrence of a triple.

### 4.4.5 Special Values (System ID Constants)

In order to start a new database, it is necessary to predefine several IDs

IDs for types:
EntType, LexType, EntEntType, EntLexType

IDs for system sets:
SetSet, SetNameSet, RelSet, RelNameSet

IDs for special Relations:
SetNameRel, SetTypeRel, RelNameRel, FromSetRel, ToSetRel, RelTypeRel

IDs for various additional constants:
LowestSystemId, "None", "Any", HasEnt

## 4.5 Interfaces

### 4.5.1 The Programming Interface

Access to the database is through a set of operations that together make up the 'TD' (triple datastore) programming interface. The interface is not intended for end users, but provides a clean interface into the database system upon which graphical front-ends and so on can be built. The operations provide the following functions:

| | |
|---|---|
| Database operations | Create a new database<br>Delete a database<br>Open a database for processing<br>Close a database when finished |
| Set operations (on both entity and lexical sets) | Add a new set<br>Delete a set<br>Find a set ID<br>Find a set name |
| Operations on members of lexical and entity sets | Add a new member<br>Delete a member |
| Operations involving relations between sets | Add/delete a new entity-entity relation<br>Add/delete a new entity-lexical relation<br>Find an entity-entity relation ID / name<br>Find an entity-lexical relation ID / name<br>Find ID of to_set / from_set<br>Find set name of to_set / from_set |
| Operations to add or delete connections between members of sets | Add/delete connection between two entities<br>Add/delete connection between entity & lexical<br>Add lexical value and connection |
| Cursor operations | Various operations – see Appendix A |

**Figure 4.3** TD Operations

These operations are fully described in Appendix A.

### 4.5.2 End-User Interfaces

*Section 4.5.2 describes four different interfaces that were developed by students under my supervision while I was teaching at the University of Southampton.*

To complement the triple store implementation, a variety of user interfaces has also been constructed, demonstrating the versatility and validity of the triple store. These included graphical interfaces, a web-based interface, and an SQL interface. Brief descriptions of these follow.

#### 4.5.2.1 General Purpose Interface

This can be used to assemble a database from scratch. Using the buttons on the toolbar, or using menus, the user can

- Create a database
- Add sets
- Add relations
- Add connections
- Add data
- Obtain various views of the contents of the database.

**Figure 4.4** General Purpose Interface

This interface (Figure 4.4) demonstrated that a Windows front-end could be added to the triple store code, using C++ and an object-oriented approach. A full description is given in [Cjs98].

#### 4.5.2.2 Interface to support recruitment agency application

This was a full-scale application based on a real system. The ER diagram in Figure 4.5 shows the database in a conventional view. Figure 4.6 shows how this is translated into a binary relational database. This interface is described in [Ejs99].

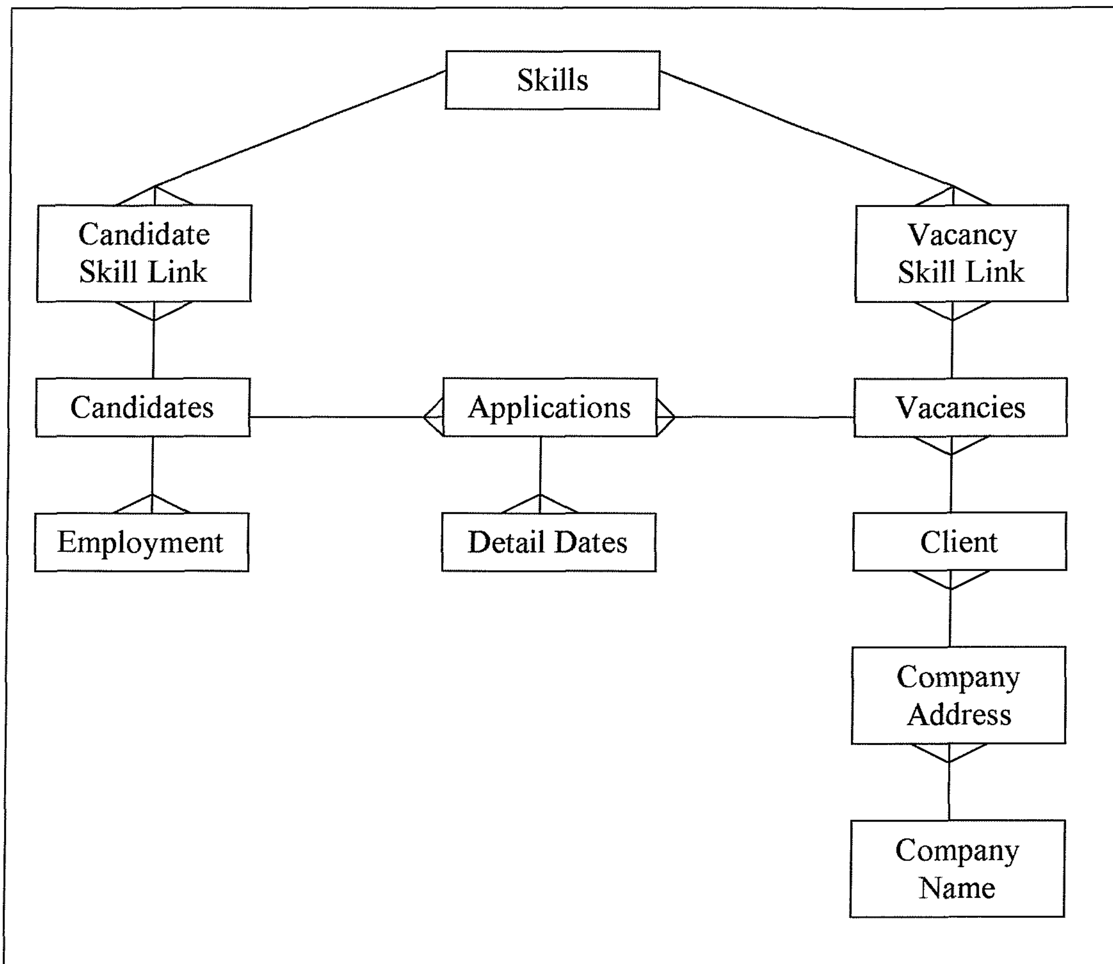
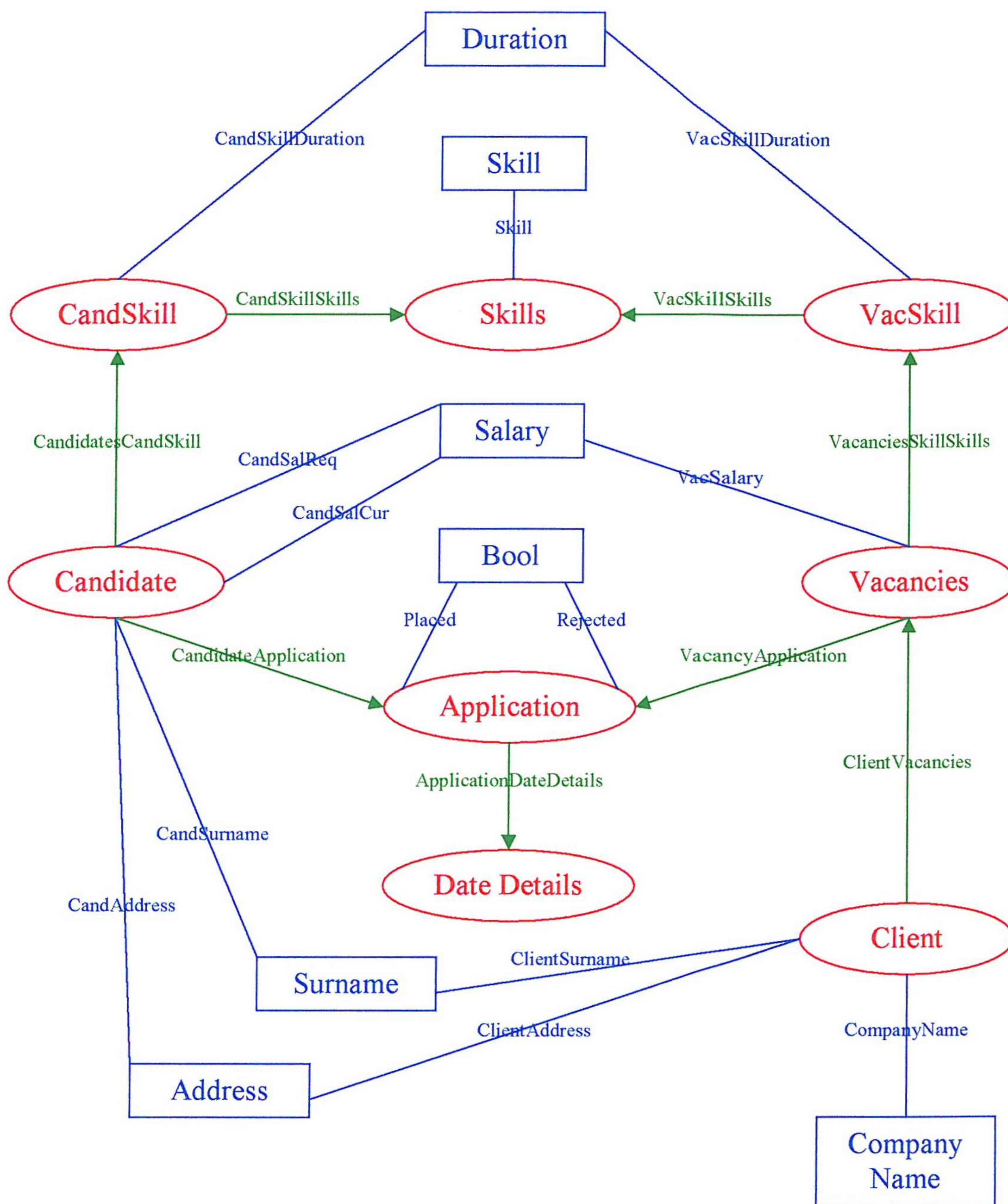



**Figure 4.5** ER Diagram for recruitment agency

The application records job vacancies supplied by clients, along with the skills required, shown on the right-hand side of the diagram. It also records candidates applying for jobs, with the skills they are offering on the left. The application matches candidates to vacancies, and records job applications made.

The following diagram shows the way in which the Triple Store holds the above data structure.




Red Entity
Green Entity Entity Relation
Blue Lexical Entity and Lexical Relation

**Figure 4.6** Data model for recruitment agency

Figure 4.7 shows a screen from the application for viewing a candidate's details, demonstrating that a conventional user interface can be developed for an application based on a triple store database.

**Figure 4.7** Recruitment application

#### 4.5.2.3 Data Explorer Web Interface

One of the features of a binary relational database is the ability to follow links between instances of data values in the database to discover whether there are connections between them. Indeed, researchers at Birkbeck College [Tristarp] have developed applications for clients, taking advantage of this. The web interface shown here [Gje00] was developed at Southampton and was designed to support such 'data exploration' through the triple store database.

From a given starting point, such as that shown in Figure 4.8, the user can click on items as indicated to track down connections. For example, one could click on the name field to find any other references to the person selected, in this case, Mrs G Wood.

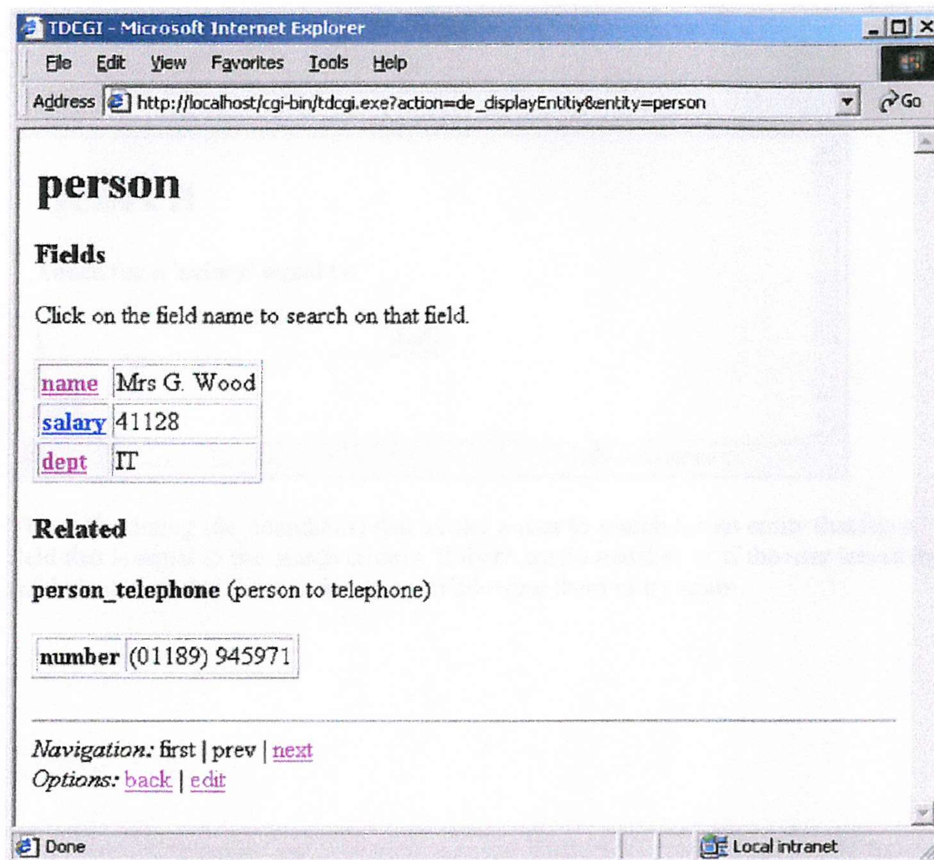



**Figure 4.8** Web Interface - Data Explorer

#### 4.5.2.4 SQL Interface

The last interface described here [Cwa00] was designed to explore the possibility of creating an n-ary relational database view of the data in a binary relational database. The aim was to design and build a text-based SQL interface. The application comprised three sections:

- the parsing layer to collect and validate input from the user
- the relational layer to implement the relational database model
- the binary relational layer concerned with interfacing to the underlying triple store

The final system was successfully able to create relational tables, insert data into those tables and query the tables to retrieve the data, with all data being held in a triple store.

## *4.6 Preliminary Performance Comparison*

To gain reassurance that the performance of the database was at least acceptable, a short study was conducted to provide a 'sanity check' [Hus99]. The same database was constructed in the triple store database and in Access, which happened to be readily available. In order to provide a meaningful comparison, programs were written to access the programming interface of the triple store database, and were also written to retrieve data directly from the underlying Jet engine in Access, so bypassing the Access front end.

The study was conducted against an earlier version of the triple store database, which had not yet had the index added to the lexical store. Nevertheless, the results were satisfactory. The triple store was faster on some queries and Access was faster on others, but both were in the same 'ball park'. The results confirmed that the triple store performance is certainly comparable with other databases, and may well be faster when fully developed. In due course, a further comparison should be performed against a more substantial competitor.

## *4.7 Other Aspects*

### 4.7.1 Locking and Robust Cursors

The present implementation includes complete physical locking at the block level, described below in Section 4.7.2. The largest amount of data that is locked at one time is one path through the index (a small number of blocks) together with the target block. This is adequate for current purposes, but to support multiple users, a stronger approach is needed. It is intended to explore the use of predicate locking for this purpose, and this is discussed further in Chapter 6.

A mechanism is already in place to support robust cursors. A cursor may be used to find the position of data within a block. If unrelated changes are then made to the block, causing the position of the data to change, the cursor will still return the correct data should it be required again. To achieve this, a version count is maintained in the block, which is recorded in the cursor. If the cursor is used again, a check is made to see

whether the version count in the cursor matches the version count in the block. If it does not, then changes have been made, and the index is used to re-locate the required data item. This applies to changes in the leaf and index blocks.

### 4.7.2 Caching and Storing to Disk

All data is stored to disk, with a caching mechanism within the database to optimize performance. The cache size may be selected at start up, in terms of the number of data blocks to be held. One aspect of the performance work has been to understand the best values for the block size and for the cache size. This is described in Chapter 5.

Each 'slot' in the cache can hold one block of data. A 'status table' holds a record for each cache slot. The contents of each record in the table are shown in Figure 4.9.

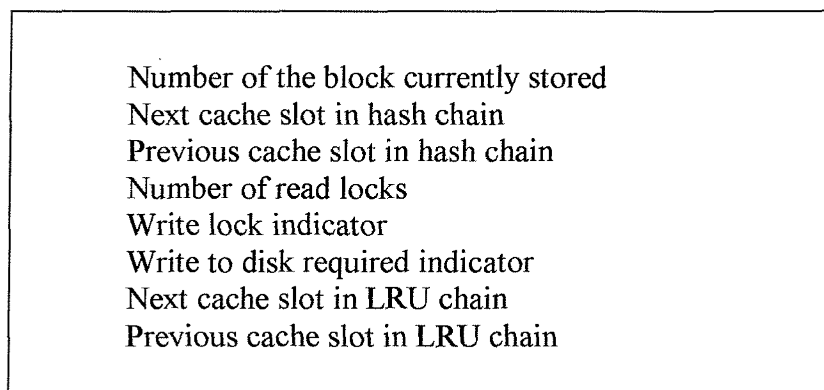



**Figure 4.9** Status table record contents

A division/remainder hashing algorithm is used to locate blocks stored in the cache. The remainder is used as the index into a separate hash table, which holds the position in the status table of the start of a relatively short chain of blocks which all have the same hash result. This level of indirection ensures that the usage of the cache is independent of the hash number, as well as providing flexibility and improving performance.

When access to a block is required, the block number is hashed, and the hash chain can quickly be followed through the status table to see if the block is in the cache, or needs to be fetched from disk, thus avoiding the need to scan the whole status table looking for a block-number match. (A large cache could contain thousands of blocks.) When a new block is brought into the cache, a slot will be allocated from the 'free chain' (see below).

The slot in which the new block is stored is added to the front of the appropriate hash chain.

The status table keeps track of the current usage of the block in each slot. Whenever a block is accessed, a read or write lock will be placed on it. Multiple read locks are permitted, and a count is kept of the number active. When a block is released, any lock held will be released. A read lock count will be decremented, a write lock will be released, and if the block needs to be written to disk, the 'write required' indicator is set. This mechanism ensures that active blocks are not removed from the cache. When a flush request is sent to the cache, all blocks with outstanding write indicators set are written to disk.

The status table also holds a 'free chain' of all unused and 'free', (i.e. unlocked) blocks. If there are no unused slots in the cache, the least recently used block with no locks outstanding will be discarded and the slot re-used. If the block has the write indicator set, it will be flushed to disk first. When a slot is re-used, it is removed from its old hash chain and added to the new one.

## *4.8 Discussion of Alternative Approaches*

### 4.8.1 Identifiers

There are various strategies that could be adopted for implementing the Triple Store. We have chosen the identifiers to have two parts - an identifier for the set name (e.g. PERSON) and an identifier for the individual instance within the set (eg id1). However, an alternative would be to have just the ID and have additional 'IsMemberOf' relations saying which ID belonged to which set. This would make for shorter, less complex IDs, but one would then need to ensure that IDs were unique across the whole database rather than just within the set, and many more lines would be needed in the Triple Store.

### 4.8.2 Relations

Another debate concerns whether to endow the relation itself with the additional property of linking two specific sets. For example, one could say that "the relation PTel always links the sets PERSON and TEL". This is the approach that has been adopted so far.

Before adopting this strategy, however, one must consider relations such as 'HasValueOf' or 'IsOwnerOf'. One might use HasValueOf to link a set of computer equipment with its value in pounds:

COMPEQ_id1 HasValueOf POUNDVALUE_id1.

One might want to use the same relation to link furniture items with their value:

FURN_id1 HasValueOf POUNDVALUE_id2.

Similarly, one might want to use IsOwnerOf for different purposes:

PERSON_id1 IsOwnerOf COMPEQ_id3
PERSON_id1 IsOwnerOf FURN_id4

If one endows such relations with the property of linking specific sets, then different relations will be needed for each of the above cases: 'EquipHasValueOf', 'FurnHasValueOf', and so on. If one wanted to establish the value of all items owned by PERSON_id1, one could not simply follow all of the appropriate IsOwnerOf Relations followed by the HasValueOf relations, but would have to issue a number of specific enquiries. In addition to such practical considerations, one must recognize that the Semantic notion of HasValueOf or IsOwnerOf appears to be exactly the same in all cases and it might seem preferable to use the same relation to represent the same thing.

One way to implement this approach would be to maintain full identifiers throughout the triple store. Other ways would involve storing additional information about relations in the triple store. However, if the relation implies the from-set and to-set, then the triple store can be significantly reduced in size.

### 4.8.3 Sets are Disjoint

Rule 2 implies that sets are disjoint. Is this a constraint, either in the database or in the real world? The constraint implies that the database designer will have to divide his world up into sets, which is indeed the normal approach to database design. In the real world, however, many things belong to more than one set, so that this is not really satisfactory.

The possibility of using an 'IsA' relationship was considered, to relate an object to the one or more sets to which it belonged, the objects being held as members of a set of the descriptions of all entities in the universe. One would need to allow m:n mappings from entity IDs via IsA's to the description set. Entity ID could then be a member of as many sets as desired. However, to start with at least, the simplifying assumption is being made that entities will be members of one and only one set. Most problems can still be solved by searching on attributes. For example, the set of all photographers can be found by searching the set of people for those with the skill 'photographer'.

A further point arises. Relationships need to know which set they are coming from and which they are going to. This is much more straightforward to implement if there are distinct sets for each entity type. One could track back through IsA's to determine set membership, but this would lead to more complex code and longer pathlengths.

### 4.8.4 Mappings

At the moment, it appears that the decision to allow only m:1 and 1:1 mappings leads to a satisfactory solution in all cases. However, this has not been tested rigorously, and further work will be needed to demonstrate that all four mappings are not actually needed. The utility of introducing an option to make a mapping mandatory also needs consideration. For example, there might be a mandatory 1:1 mapping to ensure that each person was allocated a personnel number.

# 5 Optimizing Data Storage for Performance

The preceding chapters have described how a binary relational database based on a triple store was built, using an object-oriented approach. Several end-user interfaces were also developed, which demonstrated that conventional paradigms could be used above the triple store. This in itself was an interesting insight, as it might have been thought that a non-standard database implies non-standard end-user interfaces. Having established, therefore, that the database was capable of supporting standard applications, as well as less usual ones such as the Data Explorer, it was then appropriate to carry out further investigation into the performance of the database, to see what more general conclusions could be drawn.

One alternative at this stage would have been to code various versions of the triple store database, and then conduct performance measurements. However, this would have been extremely time consuming, and could have led to a considerable amount of wasted effort. The method adopted was therefore to build a model, using an innovative approach, to explore two particular issues.

The first issue, which is of importance to the current implementation, was to discover the effect of using more than one sort order to hold the triples in the triple store. Did the benefit of storing more than one sort order outweigh the cost, and if so, which sort orders should be held?

The second issue is of significance to all database management systems, and concerns compression. To what extent can performance gains be made by compressing the data, and in particular, the non-index data, in a database? This question has become a topic of interest recently, as has already been mentioned in Chapter 2.

In developing the model, the approach taken was to use the facilities provided by a spreadsheet. In this chapter, the model is described, and then the results of the two investigations are presented.

## 5.1 The Model

Modelling has been applied to all aspects of computer technology from microprocessors [Rei98], through I/O Subsystems [Gan98] to cache assignment in databases [Levy96]. The use of a spreadsheet has also been reported in [Bond96] to reduce analysis and design time by comparing efficiencies of converters in power electronic circuits. It was decided to attempt this approach for the present project, to see whether the same benefits could be obtained.

### 5.1.1 Summary of the model

The performance model was constructed around operations at the 'TD' programming interface, at which commands are submitted to the database to enter or retrieve entities and their attributes to or from the database. Specific applications may be developed within the model by assembling sequences of the operations, and the model is then used to predict behaviour as various parameters are altered. The first area of application was to determine the order or orders in which entries in the triple store should be sorted.

### 5.1.2 Extent and Limitations of the Model

The object of the model was to predict the behaviour of the triple store database under construction as a guide to continuing design and development. The operating system obviously caches data underneath its own read/write interface, so for the model, block retrieval times were determined empirically, as described later. The present model handles the reading of data only, not updating or deletion, which will follow.

The intention was to develop a valid model that could be used to aid in the design process. Various characteristics of the machine on which the database is running must be determined in order to calibrate the model for use in predictive work. A further study could involve extending the model to predict performance on a variety of machines, but this was not the current aim.

A secondary aim of this work was to demonstrate that effective modelling can be achieved relatively economically by using, as far as possible, the standard spreadsheet facilities provided by a spreadsheet, in this case, Microsoft Excel.

## 5.1.3 Key Aspects of the DBMS being modelled

### 5.1.3.1 The cache

Within the database, two caches are maintained, one for triples and one for lexicals. In both the database and in the model, the block size being used and the size of the cache can be varied. Early experiments showed that it was worth maintaining these caches inside the DBMS in addition to the caching provided by the operating system. In the cache, blocks are maintained using a least-recently-used (LRU) algorithm.

### 5.1.3.2 Data store sizes

Various application parameters may be supplied to the model. These include:

ns - the number of entity sets in database
ne - the no of entities in sets
na - the no of attributes of each entity
nee - the no of relations between sets
nea - the no of entity-attribute (ea) relations per set
dbr - the no of entity-entity (ee) relations in the database

There are also constant values (Mx) related to the triples and lexicals needed to hold metadata;
MS = 3 = no of triples to describe one entity set
MA = 8 = no of triples to describe one attribute set and its entity-attribute relation
ME = 5 = no of triples to describe one entity-entity relation
ML = 1 = no of lexical entries to describe one set
MR = 2 = no of lexical entries to describe one attribute and its entity-attribute relation

MS (3) triples per entity set; MA (8) triples per attribute set; ME (5) triples per entity-entity relation; ML (1) lexical entries per set; MR (2) lexical entries per attribute.

From these, the model derives the figures needed. The number of triples in the store (nt), including the triples needed for the metadata, is given by:-

$$nt = (MS + ne + na*(MA+ne))*ns + nee*(ME+ne)$$
$$= (MS + MA*na)*ns + ME*nee + ne*(ns*(na + 1) + nee)$$

The number of lexical entries in the lexical store (nl), including the lexicals needed for the metadata, is given by:-

$$nl = dbr + ns*(ML + MR*nea + nea*ne)$$

In addition, system parameters are supplied to the model, including:-

idsize - identifier size
thead - the size of the header in a block in the triple store
lhead - the size of the header in a block in the lexical store
lsize - lexical size

A triple contains 3 ids, so the size is 3*idsize
A triple index entry contains 4 ids, so the size is 4*idsize

The user can also vary:-

tbs- the triple block size
lbs - the lexical block size
packing factors (triple: tpf, triple index: tinpf etc)
internal cache size,

Using these, the model can determine the sizes of the triple and lexical stores, the height of index trees (a B-tree index is used), likely cache occupancy etc. For the triple store:

| | |
|---|---|
| Triples/Block: | tb = tpf*tbs/(3*idsize) |
| Triple Index Entries/Block: | tieb = ((tbs-thead)*tinpf)/(4*idsize) |
| Triple Index Height: | tih = 1 + RoundUp(Ln(nt/tb)/Ln(tieb)) |

For the lexical store

| | |
|---|---|
| Lexicals/Block: | lb = lpf*lbs/lsize |
| Lexical Index Entries/Block: | lieb = ((lbs-lhead)*linpf)/(liesize) |
| Lexical Index Height: | lih = 1 + RoundUp(Ln(nl/lb)/Ln(lieb)) |

Finally, parameters are needed for the time taken to retrieve a block of data from the operating system (bget), and to carry out various processing elements, for example, the processing time to make one pass through the triple store index (tip) or to handle one triple leaf node (tlp). These were determined by calibration, described later.

Most calculations can be performed using spreadsheet formulae. For some calculations, Visual Basic code was written. Two macros were used to derive the number of index levels held in the triple and lexical caches, based on the quantity of data in the database. Two further macros were used to calculate the number of index nodes present. Standard

formulae give the height of an index tree, and from this one could calculate the maximum number of nodes that would be in the tree. However, what was required was to find the number of nodes actually being used to index the current quantity of data in the database for each of the triple and lexical stores, a value that might well be far smaller than the maximum possible number.

#### 5.1.3.3 Data retrieval

To understand how data retrieval is modelled, a central aspect of the operation of the triple store is summarized again here. The three elements that make up a triple are the Relation Id (relId or R), the Id of the Entity from which the relationship starts (fromId or F) and the Id of the Entity to which the relationship connects (toId or T). A request at the TD interface will normally contain two of the three elements (although sometimes only one element will be supplied). For example, a request might supply the relId and the fromId and require the toId to be found. This will be referred to as a request of the form RF*.

The data in the triple store was initially sorted in the order: R, F, T, and was indexed using a B-tree structure. As a consequence, a request of the form RF* could be quickly satisfied via the index, whereas a request of the form R*T could not. In this situation, the DBMS must use the index to find the start of the relation and then perform a sequential scan. There are six ways that the data could be sorted, and a major objective of the modelling exercise was to discover which of these would be the best, and whether there would be significant benefit in holding the data in more than one sort order. If more than one sort order were to be maintained, the assumption is that the DBMS would perform a simple optimization to use the best sort order for a given operation. The cost of performing basic operations (RF*, R*T, etc.) against each of the six different sort orders is calculated in the model, and a matrix holds the results.

#### 5.1.3.4 Formulae for triple store

At this stage, the measured values for basic elements are factored in:

tip - Triple index processing time
tlp - Triple leaf processing time
lip - Lexical index processing time
llp - Lexical leaf processing time
tsp - Triple scan processing time
bget- Time to retrieve one block at random from the file system

Because of the intrinsic symmetry of the situation, only a small number of formulae are actually required, which are variations of the following:

- **Index direct to unique item**

For data sorted in the order R, F, T, an operation of the form RF* can use the index to retrieve the required record from the database. If (tscl) represents the number of index levels held in the triple store cache, the formula for retrieval time is given by:

time = (ih-1)*tip+tlp+(bget*(ih-tscl))

- **Index to first in set**

For data sorted in the order R, F, T, an operation of the form R*T can use the index to reach the first record in the required set. Thereafter, the set must be scanned sequentially, looking for matching records.

time = 0.3*(bget*(ih-tscl))*(1+(nbr*(ifroom)))+ne*tsp)

where ifroom is a conditional expression, which determines how much, if any, space is left in the cache after all index levels have been cached

ifroom:- if (nbr > tscr) then ifroom = (20/tscr)
else ifroom = 0

- **Index to first in triple store, and then scan**

For data sorted in the order R, F, T, an operation of the form *FT cannot use any index. Therefore, the triple store must be scanned sequentially, looking for matching records.

tip + RoundUp(nt/tb)*bget

Similar formulae apply for the lexical store.

## 5.1.4 Calibration and Validation

### 5.1.4.1 Block retrieval times

A key parameter of the model is the block retrieval time - the time it takes to retrieve a block from the triple store or the lexical store. In order to obtain values for the time taken

to satisfy a request to the file system to retrieve a block, a small calibration program was developed to write a file of various sizes and then to measure the time taken to retrieve blocks of various sizes, both randomly and sequentially, on the system being modelled. This approach treats the file system as a 'black box'. As would be expected, the file system itself provides very significant caching; the memory available on the experimental machine was 64 MB. Part of the exercise was therefore to discover at what point file system caching became a significant factor, by ranging over varying file sizes and block sizes.

File performance was measured for sequential reads and for random reads, since the triple store software might result in either being needed. A 100 MB file was used, to reduce the effects of caching, at least for random access. The results are shown in Figures 5.1 and 5.2.

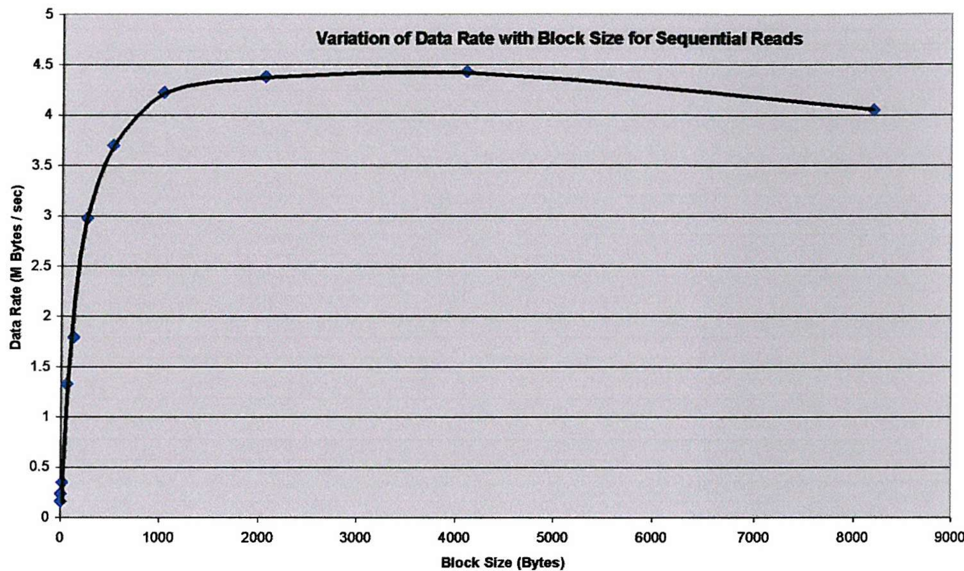



**Figure 5.1** Variation of data rate with block size for sequential reads

In Figure 5.1, it can be seen that for sequential reads, performance improves steadily up to a block size of 1 KB, and then only slightly more before starting to fall away. Most accesses to the triple store, however, will result in reading one block at random. The time to retrieve a block at random is therefore of most interest, and Figure 5.2 shows that this is lowest for a 1 KB block. Sequential scans through the triple store will result in a succession of random reads, and Figure 5.3 shows that the data rate does improve for larger block sizes. However, since it is expected that most retrieves will be for a single block, a block size of 1 KB was chosen for the present experiment.

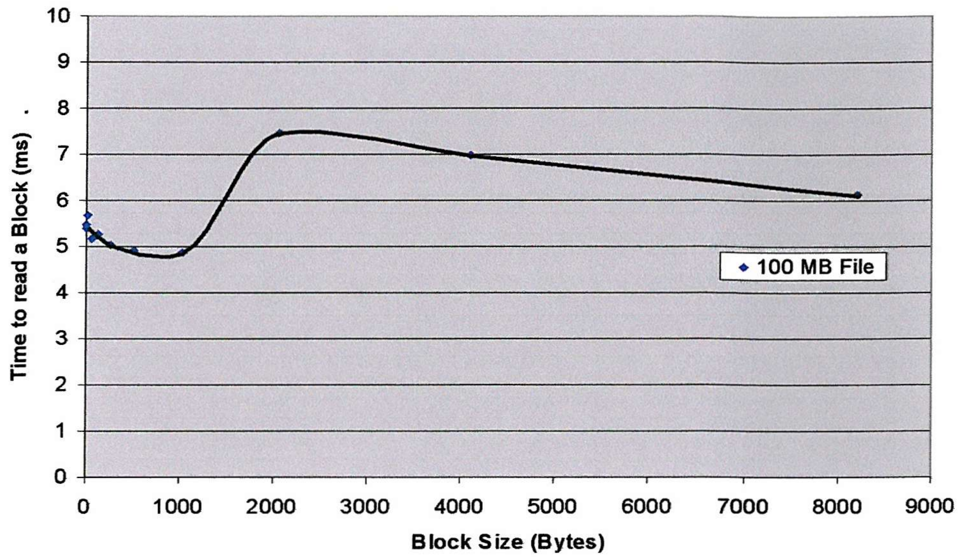



**Figure 5.2** Variation of block read time with block size for random reads

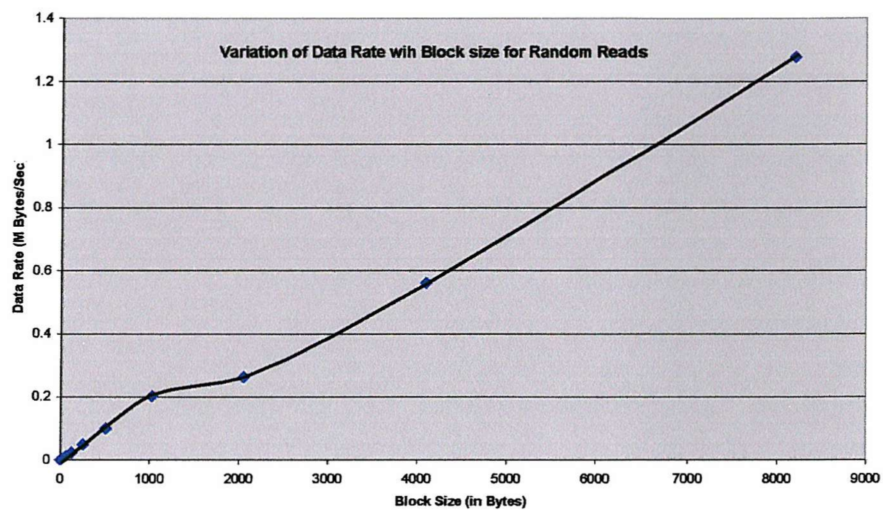



**Figure 5.3** Variation of data rate with block size for random reads

This decision was further reinforced by the findings shown in Figure 5.4, showing the variation of random read time with file size. A typical request by the triple store would be for a single triple. A block of 1 KB contains 50 or more triples (depending upon packing density). A block size of 1 KB gives a significantly faster response than a block size of 2 KB.

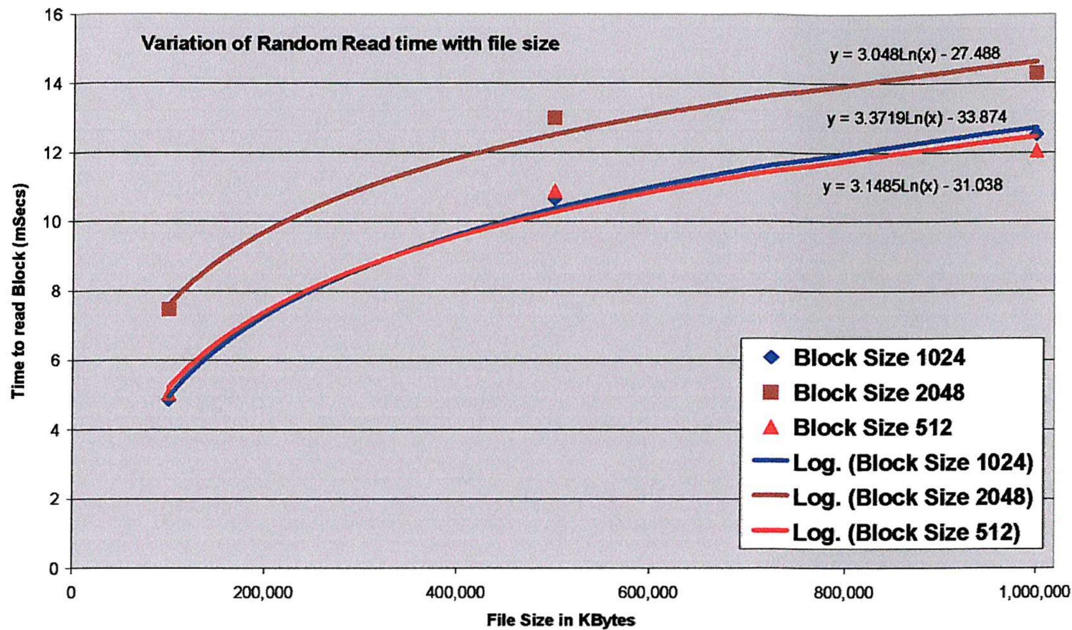



**Figure 5.4** Variation of random read time with file size

At file sizes of less than 50 MB, the whole file is effectively loaded into the system cache, and very fast retrieval times result. From this point upwards, a more regular pattern develops, which can be approximated by logarithmic formulae. It was decided to use a block size of 1 KB to give good block retrieval times, and to build data sets which took the total volume of data towards 100 MB.

#### 5.1.4.2 Processing times

Figures were obtained for processing times by measuring the performance of a sample database with given parameters. By varying the size of the test database, the size of the cache and so on, it was possible to factor out times for the various elements needed.

#### 5.1.4.3 Validation

The values obtained from the calibration were used in the model, which was then validated by comparing predicted times for queries with measured times from the database. Adjustments were made to ensure that the model reflected known performance to within 10%.

## 5.2 *Investigating Sort Orders*

### 5.2.1 Results

For this investigation, the model was calibrated against the then current implementation of the triple store database at the level of the basic operations, running on a 300 MHz Office PC, with 64 MB of memory and a 4 GB disk, with Windows 95.

A database was modelled containing two sets, Person and Telephone, linked by a many to 1 (m:1) relationship, so that each person had a telephone number, but telephone numbers could be shared. A query was set up in the model which took a person's name and returned their telephone number. The model was then run a number of times with varying parameters to investigate the effect of storing the data in different sort orders. The data given by the model was fed straight to the Excel graph facilities to produce Figure 5.5.

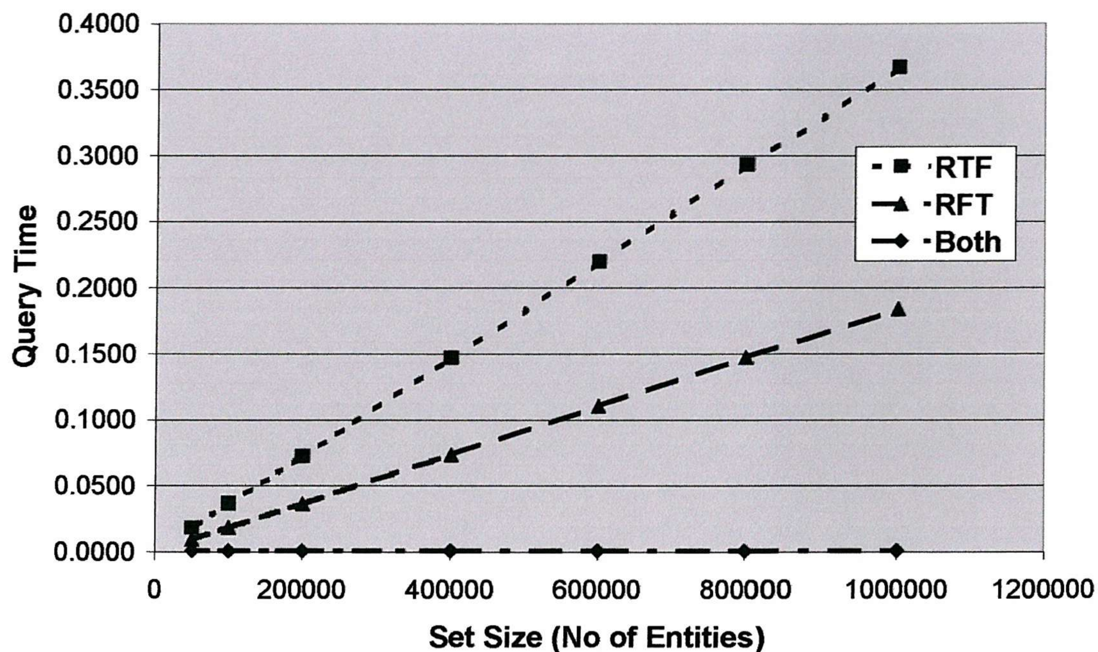



**Figure 5.5** Effect of sort order on query times

The graph shows the effect on the execution time of the query of choosing one or the other of two sort orders - RFT or RTF - to store the data in the database. It also shows what happens when the data is stored in both sort orders, with the DBMS optimizing the query to

use the most efficient sort order for retrieval. If both sort orders are included, a dramatic improvement takes place, shown by the line running just above the horizontal axis.

### 5.2.2 Discussion

The effect of holding both sort orders in the database is clear. With both orders present, the need to scan a large number of triples is removed for most queries. Accesses to the triple store when the relId is not known are comparatively rare, but are discussed later, in the context of compression.

As a result of this exercise, the triple store database was extended to include the two sort orders modelled here, and the predicted gains in performance were achieved.

## *5.3 Investigating the effect of Compression*

The subject of compression was introduced in Chapter 2, which highlighted the current interest in this topic. With the publication of recent papers by other groups, this part of the research had immediate relevance, to see how results with the triple store would compare with results in n-ary databases.

### 5.3.1 Compression in Databases

The most obvious reason to consider compression in a database context might seem to be to reduce the space required on disk. However, as disk space becomes rapidly less expensive, this is no longer such an important concern. The more important issue is to see whether the processing time for queries can be reduced by limiting the amount of data that needs to be read from disk to satisfy the query. By compressing data, can the number of blocks to be read be reduced?

Speed-up can come from reducing the number of disk I/Os, (as long as the CPU cost of achieving this is not too high) and frequently the only way to do this is by reducing the number of accesses required in traversing the index. The height of the index tree is given by a logarithmic formula:-

$$H = \frac{Ln(RBlkNum)}{Ln(INum)}$$

where H is the height of the tree, RBlkNum is the number of blocks containing data records, and INum is the number of index entries/block. In other words, there is an exponential relationship between H and both RBlkNum and INum.

One option is therefore to increase INum by compressing index entries, which is the route taken in many databases today. The second option, in which we are interested, is to decrease RBlkNum, by compressing the data itself. In order to reduce the height of the index tree by one, and thus eliminate one disk I/O, we could calculate

$$H'' - H' = \frac{Ln(RBlkNum'')}{Ln(INum'')} - \frac{Ln(RBlkNum')}{Ln(INum')} = 1$$

If we assume that INum, the degree of index compression, is the same in both cases, this simplifies to

$$Ln(RBlkNum'') - Ln(RBlkNum') = Ln(INum)$$

or

$$\frac{RBlkNum''}{RBlkNum'} = INum$$

So if the number of index entries per block were, say, 100 (a relatively low figure), then to achieve a consistent performance improvement by reducing the number of disk accesses by one for all database sizes, a compression factor of over 100 is needed, a fairly aggressive target!

This sort of analysis might lead one to abandon interest in data compression immediately, but in fact things are not quite so simple, as the following work will show. Nevertheless, the basic facts above should be borne in mind and will be discussed later.

### 5.3.2 Towards a Compression Algorithm

In the triple store, sorting ensures that the first part of the triple will be repeated for successive entries, which immediately suggests scope for compression. Each entry in the

triple store contains three parts: the identity of the relationship (relId), the identity of the entity that the relationship runs from (fromId) and the identity of the entity that the relationship runs to (toId). The triples are stored in sorted order in two ways: <relId, fromId, toId> and <relId, toId, fromId>. Each logical triple is therefore actually stored twice, and query processing is optimized to use the appropriate sort order depending on the search criteria. As entity sets increase in size, there are increasing numbers of triples for each relationship type.

The three identities are each currently represented by a 4-byte integer, which gives a symmetrical implementation. The triple store is accessed by means of a B-Tree type of index. While compression in indexes can be lossy, (if index entries are over-compressed, the situation can be recovered by retrieving additional data blocks), in the triple store itself any algorithm must not lose information.

#### 5.3.2.1 The scope for compression in the triple store

Most queries applied to the database will result in the direct retrieval of one or a small number of triples by means of the index. The only queries where this is not the case, and a range of triples is retrieved in sequence, are where the database is being searched to perform a join on a non-key field (in n-ary terms). The DBMS contains its own cache, and the size of this will affect the number of blocks that must be read from the disk. Cache size and block size are parameters in the performance model. If the triples can be reduced in size, more triples can be held in a block. The size of the index is therefore reduced, and this is also modelled.

Two observations are worth making at this point:

1) The number of different relIds in a given database is quite small. In the database described below, there are fewer than one hundred different relIds. The ID allocation algorithm is designed to pack numbers into as few low-order bytes as possible, and it is likely that there will be 'spare' bytes at the start of the relId that are never used.

2) A 16 KB block can store about 1000 uncompressed triples at 70% occupancy. The triples are sorted, so with a packing density for the IDs of 50% (i.e. the IDs are allocated so that 50% of the numbers in a given range are actually used), the range of fromIds in a block could be as little as 2000 (Hex 7D0), needing only one and a half bytes. This figure is even lower if a smaller block size is used. Within one block, therefore, it is quite likely that the high order bytes will be repeated for many successive triples.

#### 5.3.2.2 Possible approaches

Two contrasting approaches were considered. The first was typified by an algorithm which made use of a 'compression byte' prefixed to the triple. The bits in the prefix are set to indicate which bytes in the present triple are repeated from the previous triple, and are therefore omitted. Application of the algorithm to a sample triple store indicated that the store could be compressed to about 60% of its original size.

However, there is a major disadvantage to this approach, which applies in some degree to compression in most databases. In order to carry out any processing, the block will need to be decompressed, as the offset of a record depends on the size of the previous records in the block. While the reduction in size potentially gives a significant reduction in I/O, the intensity of processing in the triple store, where relationships are followed from one entity to another, led to consideration of another algorithm.

The second approach was designed to permit the processing of a block in its compressed state. The principle is that once the block has been initially compressed, subsequent operations, particularly binary searches, can be performed on the block in its compressed state, *without needing to decompress it every time*, which will clearly benefit performance considerably. The algorithm used to achieve this was termed 'the block mask algorithm'.

#### 5.3.2.3 The block mask algorithm

At the beginning of each block, a mask is stored, indicating which of the twelve bytes in each triple are *not* constant throughout the block, as shown in Figure 5.6. The next record in the block contains a full triple, a 'starter record', with the values of the fixed bytes in the appropriate position. The remainder of the block stores short fixed length records

containing only the bytes that vary. Each block will contain a different mask, so that the length of the fixed length records in each block might be different.

| **Mask** | 0001 0011 0111 |
|---|---|
| **Starter Triple** | 0010 4000 5000 |
| **Subsequent Triples** | 345987 ( = 0013 4045 5987) |
| ... | 446678 ( = 0014 4046 5678) |
| ... | ... and so on ... |

**Figure 5.6** Example of the block mask algorithm

When a block is retrieved into the DBMS, it is then possible to use the mask and the starter record to reconstruct any individual triple without the need to decompress the whole block. As described above, the algorithm works in terms of bytes. A further refinement is possible to store only the bits that change, rather than whole bytes, which allows further compression to be achieved.

#### 5.3.2.4 Evaluation of algorithm

Application of this algorithm can lead to compression down to a third of the original size of the triple, or a quarter if bit level compression is being used. Triples are compressed when being placed in the triple store. For retrieval, the search string is compressed, the required triple is located in the compressed block (typically using a binary search) and the selected triple is decompressed when located. The block mask algorithm only needs a few lines of code to pick up the mask and the starter record, and then apply these to the selected triple.

There are further detailed decisions that a final implementation would require. For example, it would be possible to insist that each block contained only triples relating to one relId. This would enhance compression, and if data sets are large so that one relId spans several blocks, would lead to a worthwhile saving. For a small database, however, this could result in an unnecessary proliferation of blocks, adversely affecting the performance. This sort of refinement is beyond the current study, however.

## *5.4 Modelling the Performance Improvement due to Compression*

### 5.4.1 The Database

For this exercise, a database for a wholesaler buying in goods from a number of suppliers, and shipping smaller quantities to various customers was used. In conventional n-ary database terms, the database had 8 tables, with a number of relationships between them. The scenario assumed was that a variety of mainly OLTP transactions would be carried out, at normal volumes. All queries in the present experiments were read-only.

The 8 tables represented customers, suppliers, orders, products and so on. The average number of fields per table was taken as 10. This translates into a triple store database with 8 entity sets with 80 attribute sets, requiring 80 different entity-attribute relationships. The foreign key relationships between the tables translate into 10 entity-entity relationships. Thus 90 different relationships were required.

In considering the compression ratio achievable, it is necessary to consider the range of values for various aspects. The following discussion is in terms of a triple store sorted in the primary order, that is, on relId and fromId.

1) **RelId**'s: For this database, 90 different ids are required, plus the small number required to handle metadata. The ids for this could therefore be handled within one byte. For any database other than the smallest, however, most blocks will contain triples relating to only one relationship. The relId will therefore compress out completely, and be held only in the block mask.

2) **FromId**'s: Following the discussion in 5.3.2.1 above, 2 bytes will be needed, which gives a range of 64k for the values of the ids in one 16 KB block. (If a smaller block size is used, the range is reduced. However, the greater compression is not significant unless very small blocks are used, and the increase in processing then outweighs the benefit. A 16 KB block size was used throughout this series of experiments.)

3) **ToId**'s: If the database is sorted on the first two fields, then the values in the toIds will be randomly scattered across the range for each set. For a database up to one million triples (which corresponds to about 50,000 entities per set or 25 MB of actual data), 2 bytes will suffice to cover the range of ids; for a database up to 200 million triples and beyond (about 1 million entities per set, 500 MB), 3 bytes will be needed.

The 12 bytes required before compression can therefore be reduced to 4 or 5 bytes after compression for this scenario. If the compression is carried down to the bit level, then the fromId could be held in 12 bits, and the smaller ranges for the toId could also be held in 12 bits, so the compressed triple could then be just 3 bytes.

In the direction fromId to toId, each triple captures one instance of a m:1 relationship, so that when the triple store is sorted in the primary order, the third field will not have any particular sequence, as reflected above. In the inverse sorted triple store, the order is relId, toId, fromId, which represents the relationships in the 1:m order. Successive triples may now have identical fields in both the relId and the toId, and the fromId will be in sorted order, so that triples can be further compressed. To model this, however, would require more detailed examination of the distribution of data in the various domains, and this was not deemed appropriate to the present level of analysis.

### 5.4.2 Establishing the Model

The size of the cache has a critical impact on performance. As the cache size increases, more levels of index and more data records can be held in the cache, and the overall performance will improve. Cache size was therefore varied to see the effect of this as it interacts with compression.

The vast majority of normal queries involve searches where the relId is known and either the fromId or the toId is also known. In either of these cases, the blocks can be accessed directly through the index, if both sort orders are held (RFT and RTF). The main interest is therefore in the retrieval time for such queries. Sequential access to the triple store is required only for a query where, in traditional RDB terms, there is no foreign key linking two tables, such as might be used in some decision support enquiries. An example in the

Wholesale database would be "Find me the suppliers and customers who share a postcode". In this case, all or part of the triple store has to be scanned looking for matches. Compression will obviously speed these searches, and this was also considered.

## 5.5 Results of Compression Investigation

### 5.5.1 Direct Access

The model was run for the wholesale database. The cache size was varied, and in each case, results were recorded for various sizes of database, both with and without compression. Figures 5.7 and 5.8 show the results for two different cache sizes. The graphs show the average number of disk accesses required for the retrieval of a triple. A database operation will often require a number of triples to be retrieved, so that variations in the number of disk accesses will be evened out, and the average is a useful figure to work with. The complex interaction between index size, database size and cache size yields local variations, such as that in Figure 5.8, where both the first two points for the compressed database show the database almost entirely in the cache, but there is a broad similarity in the results.

The effect of increasing the size of the cache by a factor of 4 can be seen in the reduction of the number of accesses by a half to three quarters of one access, depending on the size of the database. Increasing the cache size would be expected to improve performance, and the model helps quantify the degree of improvement.

The particular interest, however, is in the effect of compression. Each graph shows the effect of this, which is to reduce the number of accesses by a significant amount ranging from a quarter to three quarters of an access. This leads to an improvement by a factor of almost two in smaller databases, dropping to1.25 in large databases. This result corresponds to the OLTP situation, where each query looks for a record which may be unrelated to the previous one, and stands in contrast to the conclusions drawn by Westmann et al [Wes00], who do not expect compression to improve the performance of OLTP-style applications.

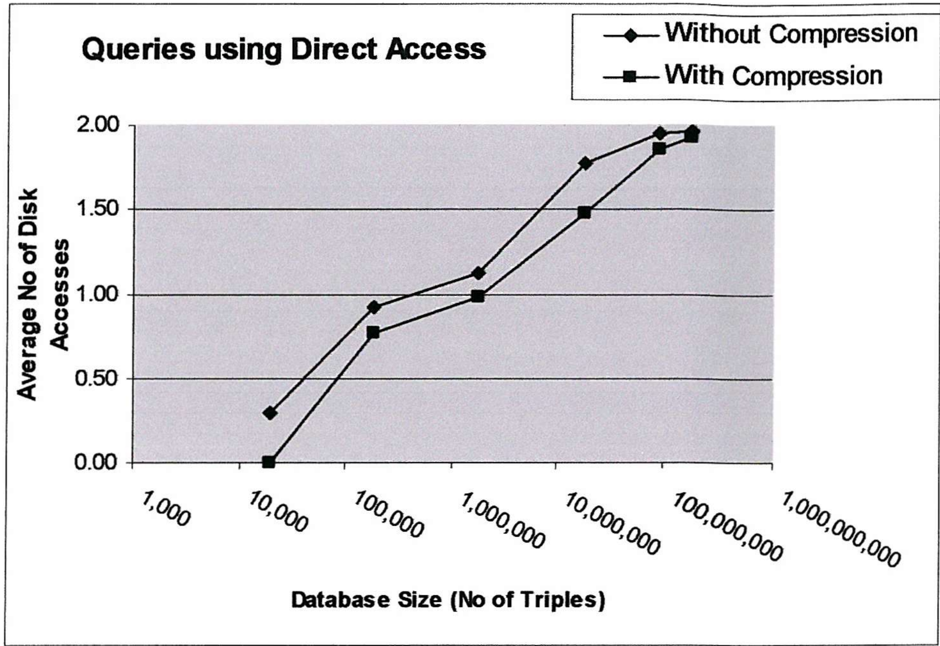



**Figure 5.7** Triple retrieval time with 256 kilobyte cache

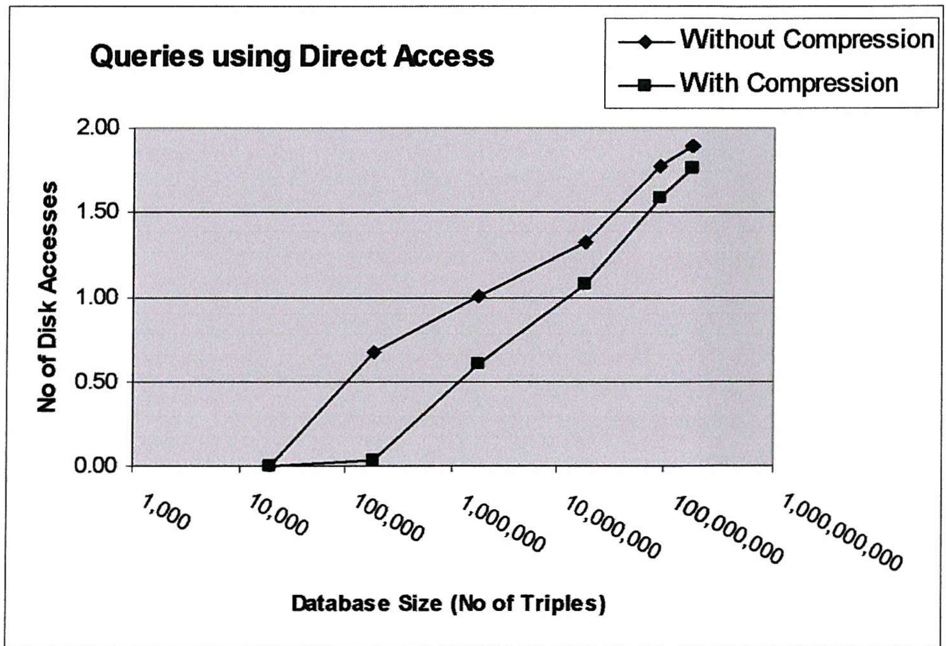



**Figure 5.8** Triple retrieval time with 1 megabyte cache

### 5.5.2 Sequential Access

For queries which do not involve a significant degree of index access, then compression produces a straightforward benefit. Each retrieved block contains more triples, in direct proportion to the compression ratio, and the model confirms this. One therefore sees an

improvement of 2:1 or better, and this is very much in line with the results from Westmann et al and Chen et al [Chen01], which both deal with decision support situations.

### 5.5.3 Discussion of Compression Results

The operations that would be carried out by joins in a conventional database are replaced by operations in the triple store, so that any reduction in the number of accesses has a direct effect on performance, whether for a single query or for a sequence of related operations. The block-mask algorithm permits processing to be carried out on compressed data, yielding a very efficient join mechanism. The effect of this has been modelled, and shown to produce significant benefit.

In the case of sequential operations which would be needed for decision support queries, the results obtained demonstrate an improvement by a factor of two. However, it has also been shown that this approach would benefit OLTP queries, giving a reduction in the number of disk accesses by a factor in the range of 1.25 to 2.

The conclusions to be drawn are considered in the final chapter.

# 6 Conclusions and Discussion

This chapter is divided into four parts. Section 6.1 draws the threads together on the effect of compression on performance, where this research has led to some interesting discoveries. In Section 6.2, the benefits of approaching modelling by using a spreadsheet are weighed. Attention then turns, in Section 6.3, to the achievements of the present implementation of the triple store database, and further developments are considered. Finally, Section 6.4 addresses the question "Could the future be binary relational?"

## *6.1 Effect of Compression on Performance*

The result of particular interest is the impact of compressing the non-index information held in the triple store. Previous research, discussed in Chapter 2, has tended to suggest that the cost of compressing and decompressing non-index data accessed randomly, as in an OLTP application, outweighs the benefit of compressing data to reduce the amount of disk access. What has been shown here is that, with a suitable algorithm, the processing cost can be contained, and database access times can be reduced, with a reduction in the number of disk accesses by a factor in the range of 1.25 to 2.

How does this compare with the results in [Wes00] and [Chen01]? They both dealt with decision support databases, in an n-ary DBMS, with Westmann et al seeing performance improve by a factor of 2, and Chen et al suggesting improvement by a factor of up to 10. For the decision support scenario, the triple store can certainly match the lower figure, and there are ways to improve this further, which are discussed below. However, neither Westmann et al nor Chen et al present results for OLTP, but Westmann et al suggest there would be no improvement here, partly because of the high cost of insertion using their approach. As long as an n-ary architecture is adhered to, these conclusions seem very reasonable.

What has been demonstrated is that by using a different architecture, it is indeed possible to use compression to speed up OLTP queries. With an n-ary database, the approaches taken are to compress different attributes in different ways, and then enhance the other

parts of the DBMS, especially the optimizer and the execution engine to deal with all the various possibilities. With the triple store, one compression algorithm is needed, and vitally, the algorithm developed then allows processing to proceed without the need, in general, to decompress triples. A very uniform implementation thus results.

OLTP applications vary widely in the ratio of reading versus updating the database. However, most transactions involve reading data initially to present data to the user (customer information, flight details and so on) and then at the end of a transaction some data may be written back to the database. Retrieval therefore usually constitutes at least 50% of the activity, and often much more, with insertions or changes making up the balance. If the cost of compression on insertion is high, as in the Westmann et al approach where specially formatted tuples are developed, even a small proportion of insertions will clearly be a problem. However, in the triple store, the cost of compression on insertions is not high, due to the nature of the algorithm and the integration with the DBMS. It is perhaps not surprising, therefore, that a different result will be obtained.

### 6.1.1 Further Compression in the Triple Store

At present, the model takes no account of locality of reference, so is actually unduly pessimistic. One of the advantages of fully decomposing data in the current implementation is that related items will be stored in close proximity, so that data is automatically clustered. This is because the whole of a binary relation is stored together in the triple store. In practice, therefore, it is expected that the results would be better than predicted by the model.

Further work should certainly include consideration of the additional effect of compression on the indexes. The uniformity of the implementation means that the same code is used to handle both the blocks in the triple store and in the index to the triple store. Any compression algorithm will therefore benefit both, and a further modelling exercise should capture the effect of this.

There is also the possibility of extending the degree of compression. The current assumption is that all data domains are large, but in practice, some are quite small. In the

extreme case of a binary domain (e.g. Male, Female), compression down to one bit per triple is possible, as follows. If a block contains one relation, and if fromIds are densely packed, then the initial fromId can be held in the block header, as well as the relId. If the rest of the block is considered as an array, each bit in the array could represent the monotonically increasing set of fromIds held in the block. Each bit could then be set to indicate whether the toId took one or other of the two possible values. This would give a compression factor of almost 100 (12 bytes down to one bit). This could be generalized and implemented on a block by block basis. If in the range of one block, the third field only uses two bits, even if the potential domain is larger, the block could be compressed to this level while retaining the higher level advantages of the triple model. This degree of compression would have a major impact on the performance of all types of queries.

The idea outlined above would lead to an automatic optimization of compression, which ties in with the idea of a self-tuning database. The triple store appears to offer significant scope for this. For instance, it would be possible to adapt the allocation of identifiers in response to the size of sets of data in order to keep number ranges compact, and the fact that data is automatically clustered has already been mentioned above. Exploration of the extent to which the database could be made self-tuning would be an interesting further avenue to explore.

## *6.2 Performance Modelling with a Spreadsheet*

The approach taken to building the performance model was to use a spreadsheet, Microsoft Excel, rather than building a model from scratch. The spreadsheet certainly provides an excellent framework within which to work, and provides many built-in routines to perform calculations. However, it soon became clear that it was necessary to be fairly sophisticated in the use of the spreadsheet, by using multiple sheets and by naming and carrying variables and values from one sheet to another, for example. It was also found necessary to code some routines which could not be achieved using spreadsheet formulae. While this is perfectly possible using VBA (Visual Basic for Applications), the novice spreadsheet user would have a further significant learning curve to travel.

The model proved versatile and easily extendable when new questions arose, and this was a major benefit. A spreadsheet provides a natural interface for holding and organizing large numbers of parameters which may then be varied. In contrast, the risk of coding a model from the ground up is that all of the requirements may not be understood at the outset, and it then becomes hard to change some of the basic assumptions. Excel also provided ready-made facilities for presenting results.

However, constructing the model was not the only part of the exercise. Calibration proved to be a time-consuming activity, as the machine on which the database was running had first to be characterized, which required long running times to load up large datasets, and then many measurements were taken using the database. This time would have to be spent, regardless of the construction of the model. The process of calibration did, though, ensure that the performance of the database as it then stood was examined systematically and became better understood than might otherwise have been the case.

The initial motivation for developing the model was to provide guidance for design decisions, and in particular, whether it would be worth adding the code to support more than one sort order in the triple store. Success was demonstrated through the investigation into sort orders using the model, and the results were subsequently corroborated when the database code was extended to support two sort orders, as indicated by the model. However, the model proved its full worth when it became the tool for conducting the wider investigation into the area of compression.

There is obviously a limit to the depth and accuracy to which it is worth developing a model. If too much time is required for model construction and calibration, it might be quicker to develop a new version of the subject of the modelling exercise, the DBMS in our case, and examine that. However, if one wants to examine a number of alternative approaches, the idea of building all of them becomes too expensive, and modelling provides the practical solution. On balance, the approach taken worked well, and permitted a model to be developed in a timely fashion which delivered the required results.

## 6.3 The Triple Store – Achievements and Further Work

This research has resulted in a new and very effective implementation of a database management system. From the outset, the intention was to keep the design as simple and pure as possible, and the architecture described in Chapter 4 achieves this. Other new aspects include the caching algorithm, the compression techniques, and the demonstration of the variety of interfaces that can be supported.

Section 6.3.1 assesses the outcome with respect to the expectations set in Chapter 1, and indicates areas for further work. Section 6.3.2 discusses the aspect of object-orientation in the triple store database. Section 6.3.3 outlines in detail one specific area where further work is essential, that of concurrency.

### 6.3.1 Demonstrating the Advantages of the Triple Store Implementation

In Chapter 1, the following advantages of basing a binary relational database on a triple store were proposed, and it is now appropriate to consider them again.

- The triple environment is essentially uniform, leading to efficiency and economy
- A considerable amount of processing can be carried out within the triple store itself, without manipulating a large number of data items
- The underlying model needs relatively simple code to access and maintain the data
- The uniformity of the triple store yields very significant compression opportunities
- The triple store also has the potential to be made completely self-tuning, which would be a significant benefit for both larger and smaller users.
- The uniform data structure is easier to spread onto multiple disks for parallelization

With regard to the first three of these points, the implementation which has been described in this thesis demonstrates their veracity. The fourth point, regarding compression, has been explored extensively, and described in the preceding chapters and sections. The fifth point concerns the extent to which the database can be made self-tuning. This was discussed in Section 6.1.1, with regard to the selection of the degree of compression in

force. It is intrinsic to the design of the implementation that there are very few parameters, and it would be a worthwhile area of study to pursue this aspect further. The final point, regarding parallelization, has proved to be beyond the scope of the present work, but is still believed to be applicable. This would be a fruitful area for further work, given the increasingly widespread availability of parallel hardware in one form or another.

### 6.3.2 The Triple Store Database and Object Orientation

The decision was made at the outset of this project to use an object-oriented approach throughout. The design reflects this, and all of the coding has been carried out in C++. As a result, it would be a very natural step to use the triple store database as the basis for an object-oriented database management system (ODBMS).

One of the distinguishing aspects of an ODBMS is that all objects are uniquely identified by an object identifier (OID) rather than using one of the data items as an identifying key as in an n-ary relational database. Each entity in the triple store database has its own identifier, so that this fundamental mechanism is in place. In addition, relationships between entities in the triple store are also dealt with entirely by the use of the identifiers, as needed in an ODBMS.

More work would be needed to develop the database into a full ODBMS, but the present implementation would provide an excellent foundation on which to build.

### 6.3.3 Concurrency Control in the Triple Store

Another area which needs to be the subject of further work is concurrency control. Some thought has been given to this, which is presented in the following section.

A problem arises in databases as soon as the database is to be used by more than one user. Multiple users may attempt to access the same data at the same time and there is the risk that data will be updated inconsistently. In order to achieve isolation, and maintain data integrity, some locking mechanism must be introduced. The first user to access a piece of data will lock the data until changes are complete, and any other user must wait until the first user unlocks the data again.

Much research has been carried out over the years on the best way to achieve such locking, as described in [Gray93] and [Bha99]. Predicate locking was first proposed by [Esw76], and in theory would give the most effective form of locking, providing isolation and dealing with the problem of 'phantoms' (see below). However, there are practical difficulties in implementation, and a trade-off has to be made between concurrency and overhead. Other techniques, most commonly some form of granular locking, are normally used. There has been intensive study of how these work out in theory and in practice. In [Sing97], for example, there is a detailed analysis of locking behaviour in three real database systems, which demonstrates the need for database administrators and designers to have an awareness of what is taking place inside the DBMS.

There remains the question of whether there are any circumstances in which it might be possible to implement a predicate locking scheme. In [Kell96], a predicate-based caching scheme for client-server databases is described, which returns to the idea. Their implementation is more optimistic than predicate locking, and is similar to precision locks. A long-term goal of the work on the triple store is to discover whether an efficient predicate locking scheme could be implemented in this environment.

#### 6.3.3.1 Predicate locking

When implementing a locking scheme, a decision has to be made about what to lock. One approach is to lock a part of the physical or logical database, depending on which part the user is trying to access. For example, one could lock the entire database, a table (or set) in the database, an individual record, or a field within a record. An alternative approach is to analyse the query being made in terms of the predicates. If the user wants to work with data relating to people with blue eyes and fair hair, there is no need to lock all of the 'people' records in the database, but only the records that satisfy the predicate < eyes = "blue" AND hair = "fair">. This is known as predicate locking.

One needs to be aware that another query using only part of the first transaction's predicate, perhaps seeking to raise the salaries of all blue-eyed boys (<eyes = "blue">), could also interfere with the first transaction, so that the predicates need to be compared carefully, but if correctly implemented, predicate locking guarantees isolation.

Predicate locking also deals with the problem of phantoms. Suppose that transaction T1 is updating the salaries of all blue-eyed boys. Using a physical locking scheme, one could lock the records relating to all of the blue-eyed boys in the database before starting the update. However, this would not then prevent a second transaction T2 starting which could add new blue-eyed boys to the database, while T1 was still executing. These new records, not locked by T1, are known as phantoms, and there are other circumstances in which phantoms can arise. Predicate locking will prevent this, as T2 would not be allowed to start since its predicate conflicts with T1.

#### 6.3.3.2 Predicate locking problems

[Gray93] describes the following three shortcomings of predicate locks

1) Execution cost. The predicate lock manager has to test for predicate satisfiability as an inner loop of the locking algorithm. Predicate satisfiability is known to be NP-complete – the best algorithms for it run in time proportional to $2^N$. This is not the sort of algorithm to put in the inner loop of another algorithm
2) Pessimism. Predicate locks are somewhat pessimistic. In other words, to ensure isolation, the mechanism may lock more of the database than is actually necessary, as it is impossible for the algorithm to comprehend constraints that exist on the actual data.
3) In general, it is difficult to discover the predicates.

#### 6.3.3.3 Predicate locking in the triple store

The intention is to implement a predicate locking scheme within the Triple Store. This will be a further investigation beyond the current thesis, but discussion is included here as indication of future direction. The triple store is a unique platform from which to investigate the issues further, because of the elegant simplicity of the design, which extends to the inclusion of all metadata within the uniform structure of the database.

One of the dangers of predicate locking is of excessive interference between locks in the index. In the case of the triple store, this might seem even more acute at first sight because there are only two indexes – one for the triple store and one for the lexical store. However, the requirements are eased by two factors.

1) Short sequential scans are often needed through the triple store to find a record. During these scans, the root of the index need not be locked for long, as it can be released once the search has started. Triples within a block are chained together, and the links can be followed, except when moving to a fresh block.
2) It is intended to implement an optimistic scheme of refining locks, to improve speed. When transaction T1 is started, locks will be held at a gross level. When transaction T2 starts, if there is a lock conflict, then T1's lock will be refined to the point where there is no conflict with T2. If this is not possible, then T2 will wait until T1 has finished, and so on.

As a result of the unified architecture, it should be possible to launch any operation at any time, including the addition or removal of sets. Updates to the dictionary (metadata) will be treated like normal transactions.

## 6.4 Could the Future be Binary Relational?

The binary relational database is an idea which continues to draw interest. This is shown not only by the number of research efforts which keep coming back to it, but also by the fact that commercial vendors find themselves drawn back to the idea, as described in Chapter 2.

One of the underlying reasons is possibly the fact that fully decomposing data leads to the ability to develop a solution with elegant simplicity. The triple store implementation, for example, permits very powerful processing with relatively few lines of code. This code also supports the metadata, and the indexing mechanisms.

It is not at all clear that the debates that were taking place in the 1970s about how to structure databases (see Section 2.1.8) were really resolved. Rather, they were overtaken by the events in which large companies started rolling out relational databases. In spite of their enormous power and widespread use, these n-ary relational databases have been found wanting in various respects, which is why work has continued on object-oriented and object-relational databases. Further consequences are that in analytical processing, databases are found to need star or snowflake schemas, in which data is deliberately de-normalized.

The present project has extended or complemented previous work in various ways. To take just two examples, Copeland and Khoshafian [Cop85, Kho96, Kho87] used only one sort order in their implementation, whereas two sort orders are employed here on the triple store, and potentially on the lexical store as well, which has been shown to be highly beneficial. Monet [Bon96] uses Binary Association Tables, which appear to introduce a great deal of redundancy, whereas in the lexical store described here, data values are held just once.

In Section 2.1.8, it was shown how the binary relational view is a very attractive approach at the logical level. The search for ways to provide an efficient implementation in software to support this logical view has led ultimately to the triple store described here. Given the simple and elegant solution that results, the question now is whether it would be a better underlying mechanism to support some of the other views of data, for example, n-ary or object-oriented. The present work has shown that it is perfectly possible to build a variety of interfaces above a triple store, and the separation of data from relationships has all sorts of benefits. As the n-ary relational bandwagon finally starts to slow down, it may well be that the day of the binary relational database is about to dawn.

# Appendix A

## *A.1 General Rules*

This appendix specifies the public interface of the Triple Store Database System. This interface includes the td classes and the cursor classes.

### A.1.1 Naming conventions

- Abbreviations
  - The following abbreviations are used in constructing names
    - Entity ent, e
    - Lexical lex, l
    - Relation rel, r

- Capitalization - Standard C++ naming convention ...
- Function names are fully qualified

### A.1.2 Data types

The following data types appear in the functions

esId Entity set id
lsId Lexical set id

### A.1.3 noId and anyId

These can stand in the position of any of the above identifier types

anyId an identifier of any id
noId an identifier of no id

- anyId: is equivalent to the * (wildcard) character used in other systems
- noId: if a function which requires identifier arguments is presented with 'noId', the function will do nothing.

### A.1.4 Statements about cursors

- All have first, next, valid members, which return 0 if cursor is validly located, and 1 if not.

### A.1.5 Database Integrity

Use of the database functions on a valid database guarantees maintenance of the following integrity constraints

All relations are based on existing sets
All entities belong to valid entity sets
All lexicals belong to valid lexical sets

## A.2 Typical Usage of td

Example

```
// A database called 'db' has been established

eId     myFromEnt;
eId     myToEnt;

// Code to set myFromEnt to some value

myToEnt = db.toEntId(myRel, myFromEnt);          // Get ToEntId
if ( !myToEntId.valid()) { ....          // Ent Id Not Valid - Handle error     }

// Carry on .....
```

## A.3 Database Operations

Note: ** beside a function denotes a user convenience function, constructed from the basic functions

**td::td** (const char* lexical_file_name, const char* triple_file_name)
*Open a database*

**td::~td** ( )
*Close a database*

void **td::info** (int lexical_info_level, int entity_info_level )
*Provide 'trace' information in cout*

| | |
|---|---|
| *_info_level = 0 | Function puts out no info |
| *_info_level = 1 | Function puts out summary info |
| *_info_level >= 2 | Function puts out more detailed info |

void **td::info** (int info_level ) **
*Provide 'trace' information for both entities and lexicals at the same level*

## A.4 Dictionary - Set Operations

esId **td::addEntSet** (const char* entity_set_name)
*Add a new entity set or locate an existing one*
Returns: id of the entity set created or found

lsId **td::addLexSet** (const char* lexical_set_name)
*Add a new lexical set or locate an existing one*
Returns: id of the lexical set created or found

void **td::delSet** (esId entity_set_id)
*Delete an entity set*

void **td::delSet** (lsId lexical_set_id)
*Delete a lexical set*

esId **td::entSetId** (const char* entity_set_name)
*Get set id for an entity set*
Returns: id of the entity set

lsId **td::lexSetId** (const char* lexical_set_name)
*Get set id for a lexical set*
Returns: id of the lexical set

*Note: In calls of the following type, the user supplies a pointer to the area where the name is to be put. For convenience, the call returns the same pointer, although strictly, this is redundant.*

char* **td::setName** (esid entity_set_id, char* entity_set_name)
*Sets the name of an entity set into entity_set_name*
Returns: the char array entity_set_name

char* **td::setName** (lsid lexical_set_id, char* lexical_set_name)
*Sets the name of a lexical set into entity_set_name*
Returns: the char array lexical_set_name

## A.5 Dictionary - Relation Operations

eerId **td::addeeRel** (const char* relation_name, esId entity_from_set_id,
esId entity_to_set_id)
*Add a new entity_entity relation between two existing sets*
Returns: id of the relation added

eerId **td::addeeRel** (const char* relation_name, const char* entity_from_set_name,
const char* entity_to_set_name )
*Add a new entity_entity relation between two existing sets*
Returns: id of the relation added

elrId **td::addelRel** (const char* relation_name, esId entity_set_id, lsId lexical_set_id)
*Add a new entity_lexical relation between two existing sets*
Returns: id of the relation added

elrId **td::addelRel** (const char* relation_name, const char* entity_set_name, const char* lexical_set_name )
*Add a new entity_lexical relation between two existing sets*
Returns: id of the relation added

void **td::deleeRel** (eerId entity_entity_relation__id)
*Delete an entity_entity relation*

void **td::deleeRel** (const char* entity_entity_relation_name )
*Delete an entity_entity relation*

void **td::delelRel** (elrId entity_lexical_relation__id)
*Delete an entity_lexical relation*

void **td::delelRel** (const char* entity_lexical_relation_name )
*Delete an entity_lexical relation*

eerid **td::eeRelId** (const char* entity_entity_relation_name )
*Get entity_entity relation id*
Returns: Entity to entity relation id

elrid **td::elRelId** (const char* entity_lexical_relation_name )
*Get entity_lexical relation id*
Returns: Entity to lexical relation id

char* **td::relName** (eerId entity_entity_relation_id, char* entity_entity_relation_name )
*Get entity_entity relation name*
Returns: the char array entity_entity_relation_name

char* **td::relName** (elrId entity_lexical_relation_id, char* entity_lexical_relation_name )
*Get entity to lexical relation name*
Returns: the char array entity_lexical_relation_name

esId **td::fromSetId** (eerId entity_entity_relation_id)
*Get from_set_id for an entity to entity relation*
Returns: Entity set id of the relation from_set

esId **td::fromSetId** (elrId entity_lexical_relation_id)
*Get from_set_id for an entity to lexical relation*
Returns: Entity set id of the relation from_set

esId **td::toSetId** (eerId entity_entity_relation_id)
*Get to_set_id for an entity to entity relation*
Returns: Entity set id of the relation to_set

lsId **td::toSetId** (elrId entity_lexical_relation_id)
*Get to_set_id for an entity to lexical relation*
Returns: Lexical set id of the relation to_set

char* **td::fromSetName** (eerId entity_entity_relation_id, char* entity_from_set_name) **
*Get from_set_name for an entity to entity relation*
Returns: the char array entity_from_set_name

char* **td::fromSetName** (elrId entity_lexical_relation_id, char* entity_from_set_name) **
*Get from_set_name for an entity to lexical relation*
Returns: the char array entity_from_set_name

char* **td::toSetName** (eerId entity_entity_relation_id, char* entity_to_set_name) **
*Get to_set_name for an entity to entity relation*
Returns: the char array entity_to_set_name

char* **td::toSetName** (elrId entity_lexical_relation_id, char* lexical_to_set_name) **
*Get to_set_name for an entity to lexical relation*
Returns: the char array lexical_to_set_name

## A.6 Data - (Entities)

eId **td::addEnt** (esId entity_set_identifier)
*Add an entity to an entity set*
Returns: id of the entity added

eId **td::addEnt** (const char* entity_set_name) **
*Add an entity to an entity set*
Returns: id of the entity added

void **td::delEnt** (esId entity_set_identifier, eId entity_identifier)
*Delete an entity from a set*

## A.7 Data - (Lexicals)

lId **td::addLex** (lsId lexical_set_identifier, const char* lexical_value)
*Add a lexical to a lexical set*
Returns: id of the lexical added

lId **td::addLex** (const char* lexical_set_name, const char* lexical_value) **
*Add a lexical to a lexical set*
Returns: id of the lexical added

void **td::delLex** (lsId lexical_set_identifier, lId lexical_identifier)
*Delete a lexical from a lexical set*

void **td::delLex** (lsId lexical_set_identifier, const char* lexical_value) **
*Delete a lexical from a lexical set*

void **td::delLex** (const char* lexical_set_name, lid lexical_identifier) **
*Delete a lexical from a lexical set*

lid **td::lexId** (lsId lexical_set_identifier, const char* lexical_value)
*Get id for a lexical value*
Returns: id of the lexical value

lid **td::lexId** (const char* lexical_set_name, const char* lexical_value) **
*Get id for a lexical value*
Returns: id of the lexical value

## A.8 Data - (Connections)

void **td::addCon** (eerId entity_entity_relation_identifier, eId entity_id, eId entity_id)
*Add a many-one connection between two entities*

void **td::addCon** (elrId entity_lexical_relation_identifier, eId entity_id, lId lexical_id)
*Add a many-one connection between an entity and a lexical*

void **td::addCon** (elrId entity_lexical_relation_identifier, eId entity_id,
const char* lexical_value) **
*Adds a lexical value to a lexical set and*
*adds a many-one connection between an entity and that lexical*

void **td::addCon** (const char* entity_lexical_relation_name, eId entity_id,
const char* lexical_value) **
*Adds a lexical value to a lexical set identified by name and*
*adds a many-one connection between an entity and that lexical*

void **td::delCon** (eerId entity_entity_relation_identifier, eId entity_id, eId entity_id)
*Deletes a many-one connection between two entities*

void **td::delCon** (elrId entity_lexical_relation_identifier, eId entity_id, lId lexical_id)
*Deletes a many-one connection between an entity and a lexical*

eId **td::toEntId** (eerId entity_entity_relation_identifier, eId entity_id)
*Returns the entity id of an entity in the 'to' set, given the relation id*
*and the entity id of the entity in the 'from' set*
Returns: An entity id

elId **td::toLexId** (elrId entity_lexical_relation_identifier, eId entity_id)
*Returns the lexical id of a lexical in the 'to' set, given the relation id*
*and the entity id of the entity in the 'from' set*
Returns: A lexical id

char* **td::toVal** (elrId entity_lexical_relation_identifier, eId entity_id, char* lexical_value )
*Returns the value of a lexical in the 'to' set, given the relation id and the entity id of the entity in the 'from' set*
Returns: the char array lexical_value

## A.9 Cursors

| **Cursors to do:** | **What they do** |
|---|---|
| entSetCursor | Move between entity sets in a database |
| lexSetCursor | Move between lexical sets in a database |
| eeRelCursor | Move between e-e relations in a database |
| elRelCursor | Move between e-l relations in a database |
| entCursor | Move between entities in a set |
| lexCursor | Move between lexicals in a set *(Returns in alphabetical order within set)* |
| entEntCursor | Move between connections in an entity-entity relation |
| entLexCursor | Move between connections in an entity-lexical relation |

All cursors include **first, next, valid** members, which return:-
0 if cursor is not validly located

**Examples**

1) To loop through a set of values:

```
entSetCursor esc(db);
for(esc.first(); esc.valid(); esc.next()) {
        // Do something
}
// Continue
```

2) To test values explicitly:

```
curVal = elTPers.first(myRel, anyId, myEntId);  // Cursor to first entity
if (curVal == 0) {
        cout << " Cursor not valid"  << endl;
        //   ... take action
}
// Continue
```

**Lexical Set Cursor**

**lexSetCursor::lexSetCursor** (td &database)

*Create a cursor to move between lexical sets in a database*

int **lexSetCursor::first ()**
*Locate the first lexical set in the database*

lsId **lexSetCursor::setId ()**
*Retrieve the id of a lexical set*
Returns: id of the lexical set that the cursor locates

int **lexSetCursor::next ()**
*Locate the next lexical set in the database*

int **lexSetCursor::valid ()**
*Returns the state of the cursor*

Other cursors follow the same pattern.

## References


[Ant96] G Antoshenkov, D Lomet and J Murray, "Order Preserving Compression", *Proc IEEE Conference on Data Engineering*, pp 655-663, New Orleans, LA, USA, 1996

[Ash68] W L Ash and E H Sibley, "TRAMP; An Interactive Associative Processor with Deductive Capabilities", *Proceedings of the ACM 23rd National Conference*, pp 144-156, Brandon/Systems Press, Princeton, NJ, 1968

[Atk87] M P Atkinson and O P Bunaman, "Types and Persistence in Database Programming Languages", ACM Computing Surveys, 19(2), pp 105-190, 1987

[Bac69] C Bachman, "Data Structure Diagrams", *Data Base (Bulletin of ACM SIGFIDET), 1:2*, March 1969

[Bha99] B Bhargava, "Concurrency Control in Databases", *IEEE Transactions on Knowledge and Data Engineering, Vol 11, No 1*, pp 3-16, 1999

[Bjo82] D Bjorner and H Lovengren, "Formalization of Database Systems and a Formal Definition of IMS", *Proceedings of the International Conference on Very Large Databases*, 1982

[Blas76] M Blasgen, R Casey and K Eswaran, "An Encoding Method for Multifield Sorting and Indexing", *Technical Report RJ 1753*, IBM Research, San Jose, CA, USA, 1976

[Bon96] P A Boncz, F Kwakkel and M L Kersten, "High Performance Traversals in Monet", *Proceedings of the British National Conference on Databases (BNCOD)*, pp 152-169, 1996

[Bon99] P A Boncz and M L Kersten, "MIL Primitives for Querying a Fragmented World", *VLDB Journal, Vol 8, No 2,* pp 101-119, 1999

[Bond96] E Bond, J Wang, WG Dunford, K Mauch, "A Simple Spreadsheet-Based Converter Performance Model Reduces Analysis and Design Time", *IEEE Industry Applications Society Annual Meeting*, Ch 355, pp 891-897, 1996

[Catt95] R G G Cattell (ed), "The Object Database Standard: ODMG-93 Release 1.2", Morgan Kaufmann, 1995

[Cha76] D Chamberlin et al, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", *IBM Journal of Research and Development 20:6,* November 1976

[Chen76] P Chen, "The Entity-Relationship Model - Towards a Unified View of Data", *ACM Transactions on Database Systems (TODS), Vol 1(1)*, ACM, January 1976

[Chen01] Z Chen, J Gehrke, F Korn, "Query Optimization in Compressed Database Systems", *ACM SIGMOD Record Vol 30, No 2*, pp 271-282, June 2001

[Cjs98] C J Smith, "Graphical User Interface for a Triple Store Database Engine", *Project Report, Department of Electronics and Computer Science, University of Southampton*, May 1998

[CODASYL] Data Description Language Journal of Development, Canadian Government Publishing Centre, 1978

[Codd70] E Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM Vol 13, No* 6, June 1970

[Codd71] E Codd, "A Database Sublanguage Founded on the Relational Calculus", *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control*, November 1971

[Codd72] E Codd, "Relational Completeness of Data Base Sublanguages", in [Rus72]

[Codd72a] E Codd, "Further Normalization of the Data Base Relational Model", in [Rus72]

[Codd74] E Codd, "Recent Investigations in Relational Database Systems", *Proceedings of the IFIP Congess*, 1974

[Com79] D Comer, "The Ubiquitous B-tree", *ACM Computing Surveys, 11(2)*, pp 121-137, 1979

[Cop85] G P Copeland and S N Khoshafian, "A Decomposition Storage Model", *Proc 1985 ACM SIGMOD International Conference on the Management of Data*, pp 268-279, ACM 1985

[Cwa00] C W Adams, "SQL Interface for a Triple Store Binary Relational Database", *Project Report, Department of Electronics and Computer Science, University of Southampton*, May 2000

[Dar96] H Darwen, "In Reply to Domains, Relations and Religious Wars", *SIGMOD Record Vol 25, No 4*, pp 6-7, December 1996

[Dat00] C J Date, Database Systems, 7th Edition, Addison-Wesley, 2000

[DB2] IBM's DB2 Web Site: http://www-3.ibm.com/software/data/db2/

[DeW90] D J DeWitt et al, "The Gamma Database Machine Project", *IEEE Transactions on Data and Knowledge Engineering 2(1),* pp 44-63, March 1990

[DMP82] Data Mapping Program – User's Guide, SB11-5340, IBM, 1982

[Elm00] R Elmasri & S B Navathe, Fundamentals of Database Systems, 3rd Edition, Addison-Wesley, 2000

[Ejs99] E Sutherland, "User Interface for a Triple Store Database", *Project Report, Department of Electronics and Computer Science, University of Southampton,* May 1999

[Esw76] K P Eswaran, J Gray, R Lorie and I L Traiger, "The Notions of Consistency and Predicate Locks in a Database System", CACM 19(11): pp 624-633, 1976

[Exc] EXcelon coporation web site: http://www.exceloncorp.com

[Feld69] J A Feldman and P D Rovner, "An ALGOL-based Associative Language", *Communications of the ACM 12, No 8,* pp 439-449, 1969

[Fitz90] J S Fitzgerald and C B Jones, "Modularizing the Formal Description of a Database System", *University of Manchester Technical Report, UMCS-90-1-1,* 1990

[Fro82] R A Frost, "Binary-Relational Storage Structures, *The Computer Journal Vol 25, No 3,* pp 358-367, 1982

[Fro86] R A Frost, "Introduction to Knowledge Base Systems", ISBN 0-00-383114-0, Collins, 1986

[Gan98] G R Ganger, "Using System-Level Models to Evaluate I/O Subsystem Designs", *IEEE Transactions on Computers, Vol 47, No 6,* pp 667-678, June 1998

[Giles82] D A Giles, "A Formal Approach to Database Design - The Triple Model", *DPhil Thesis,* Wolfson College, Oxford, 1982

[Gje00] G J Estey, "A Web-Based User Interface for the Triple Store Database", *Project Report, Department of Electronics and Computer Science, University of Southampton,* May 2000

[Gold98] J Goldstein, R Ramakrishnan and U Shaft, "Compressing relations and indexes", *Proc IEEE Conference on Data Engineering,* Orlando, FL, USA 1998

[Gra90] G Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", *Proc 1990 ACM SIGMOD International Conference on the Management of Data,* pp 102-111, ACM 1990

[Gra91] G Graefe and L Shapiro, "Data Compression and Database Performance", *Proc ACM/IEEE-CS Symposium on Applied Computing*, Kansas City, MO, USA, April 1991

[Gray93] J Gray and A Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993

[Gys89] M Gyssens, J Paradaens and D Van Gucht, "A Grammar-Based Approach Towards Unifying Hierarchical Data Models", *Proc 1989 ACM SIGMOD International Conference on the Management of Data*, ACM 1989

[Huff52] D Huffman, "A Method for the Construction of Minimum Redundancy Codes", *Proc IRE, 40(9),* pages 1098-1101, Sept 1952

[Hus99] A-T Hussain, "Triple Store versus Conventional DBMSs", *Project Report, Department of Electronics and Computer Science, University of Southampton*, May 1999

[Jag89] H Jagadish, "Incorporating Hierarchy in a Relational Model of Data", *Proc 1989 ACM SIGMOD International Conference on the Management of Data*, ACM 1989

[Kar97] K Karadimitriou, J M Tyler, "Min-Max Compression Methods for Medical Image Databases", *SIGMOD Record,* Vol 26, No 1, March 1997

[Kell96] A M Keller and J Basu, "A Predicate-Based Caching Scheme for Client-Server Based Architectures", *VLDB Journal, Vol 5, No 1*, pp 35-47, 1996

[Kho87] S Khoshafian, G Copeland, T Jagodits, H Boral and P Valduriez, "A Query Processing Strategy for the Decomposed Storage Model", *Proc 3rd International Conference on Data Engineering*, pp 636-643, IEEE 1987

[Kho93] S Khoshafian, "Object-Oriented Databases", Wiley, 1993

[Kho96] S Khoshafian and A B Baker, "Multimedia and Imaging Databases", pp 448-451, ISBN 1-55860-312-3, Morgan Kaufmann, 1996

[King90] P King, M Derakhshan, A Poulovassilis and C Small, "TriStarp – An Investigation into the Implementation and Exploitation of Binary Relational Storage Structures", *Proceedings of the British National Conference on Databases (BNCOD) 8*, pp 64-84, 1990

[Lazy] Lazy Software Web Site: http://www.lazysoft.com

[Lev67] R E Levien and M E Maron, "A Computer System for Inference Execution and Data Retrieval", *Communications of the ACM 10*, pp 715-721, 1967

[Levy96] H Levy, "The Cache Assignment Problem and Its Application to Database Buffer Management", *IEEE Transactions on Software Engineering, Vol 22, No 11*, Nov 1996

[Mar92a] J A Mariani, "Ogetto: An Object Oriented Database Layered on a Triple Store", *The Computer Journal, Vol 35, No 2*, pp 108-118, 1992

[Mar92b] J A Mariani and R Lougher, "TripleSpace: An Experiment in a 3D Graphical Interface to a Binary Relational Database", *Interacting with Computers, Vol 4, No 2*, pp 147-162, 1992

[McG77] W McGee, "The Information Management System IMS/VS, Part 1: General Structure and Operation", *IBM Systems Journal 16:2*, June 1977

[McG80] D R McGregor and J R Malone, "The FACT Database System", *Proceedings of Symposium on Research and Development in Information Retrieval, Cambridge*, Butterworths, Sevenoaks, Kent, 1980

[Mof97] A Moffatt, J Zobel, "Text Compression for Dynamic Document Databases", *IEEE Transactions on Knowledge and Data Engineering*, Vol 9, No 2, March-April 1997

[ObDb] Objectivity/DB Web Site: http://www.objectivity.com

[ObDes] Object Design Web Site, suppliers of ObjectStore: http://www.odi.com

[OCon00] S J O'Connell and N Winterbottom, "A Performance Model of Sort Orders in a Triple Store Database", *Proceedings of the Fifth International Conference on Computer Science and Informatics (CS&I 2000)*, 2000

[OCon02] S J O'Connell and N Winterbottom, "Performing Joins without Decompression in a Compressed Database System", *Submitted in Feb 2002 for consideration for publication in ACM SIGMOD Record*

[ODMG] Object Data Management Group Web Site: http://www.odmg.org

[Oracle] Oracle Web Site: http://www.oracle.com

[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*, pp 3-25, 1995

[Ram00] R Ramakrishnan & J Gehrke, Database Management Systems, McGraw Hill, 2000

[Ray95] G Ray, J Haritsa and S Seshadri, "Database Compression: A Performance Enhancement Tool", *Proc COMAD,* Pune, India, Dec 1995

[Rei98] M Reilly, J Edmondson, "Performance Simulation of an Alpha Microprocessor", *Computer, Vol 31, No 5*, pp 50-58, IEEE 1998

[Roth93] M Roth and S Van Horn, "Database Compression", *ACM SIGMOD Record, 22(3)*, pp 31-39, September 1993

[Rus72] R Rustin (ed), "Data Base Systems", Prentice-Hall, 1972.

[Sch75] H A Schmid and J R Swenson, "On the Semantics of the Relational Model", *Proceedings of the SIGMOD Conference*, San Jose, CA, USA pp 211-223, 1975

[Sen01] "Sentences DB", based on the Associative Model of Data, from Lazy Software, www.lazysoft.com

[Senko73] M E Senko, E B Altman, M M Astrahan and P L Fehder, "Data Structures and Accessing in Data-base Systems", *IBM Systems Journal 12(1)*, pp 30-93, 1973

[Senko77] M E Senko, "Data Structures and Data Accessing in Data Base Systems, Past, Present, Future", *IBM Systems Journal 16(3)*, pp 208-257, 1977

[Senko80] M E Senko, "A Query-Maintenance Language for the Data Independent Accessing Model II", *Information Systems Vol5*, pp 257-272, 1980

[Shar78] G C H Sharman and N Winterbottom, "The Data Dictionary Facilities of NDB", *Proc 4th Int. Conf on Very Large Databases (VLDB)*, pp 186-197, IEEE 1978

[Shar79] G C H Sharman and N Winterbottom, "NDB: Non-Programmer Database Facility", *IBM Technical Report TR 12.179*, IBM UK Laboratories Ltd, Hursley Park, Winchester, UK, 1979

[Shar88] G C H Sharman and N Winterbottom, "The Universal Triple Machine: a Reduced Instruction Set Repository Manager", *Proceedings of the British National Conference on Databases (BNCOD) 6,* pp189-214, 1988

[Sing97] V Singhal and A J Smith, "Analysis of Locking Behavior in Three Real Database Systems", *VLDB Journal, Vol 6, No 1*, pp 40-52, 1997

[SQLServ] Microsoft SQL Server web site: http://www.microsoft.com/sql

[Ston96] M Stonebraker with D Moore, "Object-Relational DBMSs: The Next Great Wave", Morgan Kaufmann, 1996.

[Sut95] D R Sutton and P J H King, "Incomplete Information and the Functional Data Model", *The Computer Journal, Vol 38 No 1*, pp 31-42, 1995

[Sybase] Sybase Web Site: http://www.sybase.com

[Tam] Software AG Web Site for Tamino: http:// www.softwareag.com/tamino

[Tit74] P Titman, "An Experimental Database using Binary Relations", *Data Base Management, Proceedings of the IFIP-TC-2 Working Conference*, Cargese, Corsica, Jan 1974, J W Klimbie and K L Koffeman editors, North-Holland Publishing Co, Amsterdam, 1974

[Todd76] S Todd, "The Peterlee Relational Test Vehicle", *IBM Systems Journal, 15:4*, pp 285-308, Dec 1976

[TPC95] Transaction Processing Performance Council (TPC) benchmark D (Decision Support), May 1995 www.tpc.org

[TPC99] Transaction Processing Performance Council. TPC-H benchmark www.tpc.org, 1999

[TriStarp] TriStarp Web Site: http://www.dcs.bbk.ac.uk/tristarp

[Wag73] R Wagner, "Indexing Design Considerations", *IBM Systems Journal 12(4),* pp 351-367, 1973

[Welch84] T Welch, "A Technique for High Performance Data Compression", *IEEE Computer, 17(6),* pp 8-19, June 1984

[Wes00] T Westmann, D Kossmann, S Helmer, G Moerkotte, "The Implementation and Performance of Compressed Databases", *ACM SIGMOD Record Vol 29, No 3*, pp 55-67, September 2000

[Wit87] I Witten, R Neal and J Cleary, "Arithmetic Coding for Data Compression", *Communications of the ACM 30(6),* pp 520-540, 1987

[XML] W3C site on XML: http://www.w3.org/XML

[Ziv77] J Ziv, A Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory,* 22(1), pp 337-343, 1977

[Zlo75] M Zloof, "Query by Example", *Proceedings of the National Computer Conference(NCC), published by American Federation of Information Processing Societies (AFIPS) 44,* 1975