A DATABASE SYSTEM ARCHITECTURE SUPPORTING COEXISTING QUERY LANGUAGES AND DATA MODELS

by

PEDRO E. HEPP

Ph. D.

University of Edinburgh

1983

ABSTRACT

Database technology is already recognised and increasingly used in administering and organising large bodies of data and as an aid in developing software. This thesis considers the applicability of this technology in small but potentially expanding application environments with users of varying levels of competence. A database system architecture with the following main characteristics is proposed:

1. It is based on a set of software components that facilitates the implementation and evolution of a software development environment centered on a database.

2. It enables the implementation of different user interfaces to provide adequate perceptions of the information content of the database according to the user's competence, familiarity with the system or the complexity of the processing requirements.

3. It is oriented toward databases that require moderate resources from the computer system to start an application. Personal or small-group databases are likely to benefit most from this approach.

ACKNOWLEDGEMENTS

I would like most of all to express my gratitude to my wife Mariana to whom I dedicate this thesis. I am deeply indebted for her self-sacrifice, friendship and encouragement. She, and my daughters Marianita and Josefina have helped to make of this three years an enjoyable Scottish experience.

I would also like to thank my supervisor Malcolm Atkinson, to whom I owe a great deal. His continuous friendly support, dedication and guidance have had a fundamental influence on the development of this research and on my appreciation of the role of a scientist in a university environment.

I am also indebted to many others who contributed to the progress of this research. In particular, I thank Krishna Kulkarni, Segun Owoso and Rob Procter for many useful discussions, Paul Cockshott, Ken Chisholm and George Ross for helping with the PS-algol environment, Moira Norrie for many useful comments on the content of this thesis and Simon Lin for his profound lessons on Tai Chi.

This research has greatly benefited from the PS-algol project and my thanks go to all involved in it. PS-algol is partly the result of SERC grants GRA86541, GRC21977 and GRC21960 and a grant from ICL.

My thanks go also to the Pontificia Universidad Catolica de Chile for allowing me and my family to come to the United Kingdom and providing us with financial support during the last three years, to the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom for granting me twice an award under the ORS-Fees Support Schema (ORS/81195), and to the Computer Science Department at the University of Edinburgh for providing the necessary resources to conduct this research.

Finally, I would like to thank the people of this country for enrichening my life in many important ways through their culture and institutions.

DECLARATION

I declare that this thesis was composed by myself and that the research work described was performed by myself, unless specifically stated otherwise within the context.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1 The Growing Need for Database Technology

The advantages of properly organising information has always been recognised but it has been the means of doing it that has dictated the quality of the endeavour. Database technology evolved from the need to integrate large volumes of corporate information to lower software production and maintenance costs. The efforts toward separating physical from logical data representation and the emergence of high level database query languages made them accessible to a wider, less experienced user community.

Database software is typically large and therefore expensive to acquire and maintain. However, it can be scaled down by properly identifying the levels of functionality required for smaller applications and by using adequate software development tools. It can then be cost-effective for personal or small-group databases which do not need all the functionality generally associated with large scale database software.

Other reasons for the increasing popularity of DBMS and for the large number of existing Query/Update Languages are:

- Software costs can be reduced by cutting the time required to write application programs through high-level query languages.

- Software reliability can be improved, simplifying software maintenance by using DBS as program libraries together with their documentation and specifications.

- Databases are considered essential in modern software development environments, playing an integrating and unifying role in interfacing a pool of software tools. These communicate through the database which acts as the common repository of information [NBS 81].

Increasing usability of database technology for small applications will also arise from the current trend toward more powerful and less expensive small machines.

1.2 The Small Computer Market Trend

Small applications and personal or small-group databases may prove the fastest growing database areas and therefore in need of adequate software tools. In 1982 almost 1,5M microcomputers were shipped around the world and it is expected that this figure will rise to 5M in 1986 and 11M in 1991, most of them for the business community. Also forecast, is an increasing use of 16 bit machines, significant drops in prices (20 %) and a fivefold increase in the average amount of RAM (48K -> 256 K bytes) [Blundell 83].

By 1986, 235,000 micro computers will be shipped that will typically have a 16 or 32 bit processor, floating point arithmetic, 1M byte of memory and 20M bytes of hard disk all at around $ 10,000 [Serlin 83].

The introduction of computer science courses at all levels of education will probably help increase the demand for micro computers. Another significant trend is that these machines are incorporating higher level languages such as Pascal and more powerful operating systems such as Unix.

It is therefore expected that database technology will improve its chance to play a major role in helping people organise and maintain their information, despite its quantity or the level of competence of the users.

1.3 Thesis Organization

Chapter 2 presents an overview of some of the current relational database systems. Some large systems such as System R, QBE and Ingres and some small systems such as Logiquest III are analysed against a few criteria considered important for evolutionary systems.

Chapter 3 discusses some problems in software engineering for database systems in the context of this research and then outlines a possible solution comprising an internal data model built upon a modular architecture. Prototypes, software components and internal database standards are the main topics addressed. This approach is developed in the following chapters.

Chapter 4 addresses some issues in data modeling that lead to the decision to use a subset of the extended relational model proposed by Codd [Codd 79] as the internal data model. This model is then discussed in some detail and the mappings from external models to the internal model are developed. The functional query language EFDM [Kulkarni 83] is used to demonstrate the capabilities of the chosen internal model to support different external data models.

Chapter 5 briefly presents the language PS-algol with which much of the software built for this research was implemented. The availability of a language with facilities as in PS-algol is considered very beneficial in building a database environment such as the one implemented during this research.

Chapter 6 presents the architecture of the proposed system. A number of components are described and an incremental approach to their implementation is outlined. How to extend this architecture and how to accommodate it to different application environments is also indicated.

Chapter 7 describes the database environment built to test the feasibility of the previous architectural propositions. Two relational query languages, a query monitor, a functional query language and a report generator are described.

Chapters 8 and 9 address two important aspects in evolutionary systems which become more important as the volume of data grows or the processing requirements become more complex. These aspects attempt to smooth evolution by considering the effect of efficient data structures and the feedback obtained from observing user behaviour.

Chapter 10 presents the conclusions of this research, including an assessment of the work, and some directions for further research.

Chapter 2
Overview Of Some Current Database Systems

2.1 Introduction

This chapter briefly examines a few implemented database systems, highlighting those features that are considered important in an evolutionary system. The criteria for judging evolution, and what is meant by it is also defined.

The focus is mainly on relational systems that have been implemented, have been tried by a variety of users or have been subject to controlled experiments on usability, and provide at least one interactive user-interface. Although these constraints severely limit the range of systems than can be included in this overview, they are central to the subject of this chapter.

Fortunately the most representative of the "large" relational systems, namely System R, Ingres and QBE, have been thoroughly scrutinised by several researchers and some of the conclusions arrived at are presented here. On the other hand, "small" systems, which are relatively more important than large ones in the context of this thesis, still have to make an impact in the research community and the lack of independent reports leads to only partial conclusions on them. Therefore, these conclusions should be regarded only as provisional.

Other analyses of query languages can be found in [Atkinson 81a, Cuff 79, Date 81, Dieckmann 81, Kim 79, Krass 81, Pirotte 78, Schmidt 83, Ullman 82, Vandijck 78].

2.2 Features of Interest

The following questions summarise the aims of the overview:

1. How does a system cope with quantitative evolution ?

i.e. is it possible to progress "smoothly" from small to large volumes of data ?

2. How does a system accommodate user differences and proficiency evolution ?

i.e. are suitable interfaces provided for both expert and non-expert users and can they progressively become more proficient with the system ?

2.2.1 Quantitative Evolution

To answer the first question we consider those users that start organising their data on a small scale, possibly on a personal computer, and then progressively augment their volumes of data, needing perhaps to migrate to a machine with more capacity. A "small" database is one having thousands or tens of thousands of records instead of millions [Ullman 82].

Changing computer may also be desirable to accommodate technological advances. This is an important factor for small computer users, for whom a technological advance (e.g. in storage media), together with continuing drops in price, may justify the change. To smooth this type of transition, the software will have to be portable at a reasonable cost.

The initial cost of setting up a database should be proportional to the volumes and complexity of data. Thus, a user starting an application with little data should require comparable effort by the system, both in hardware (storage and cpu cycles) and software (demands for real memory) and should only need to make an effort of learning and description commensurate with the planned task.

As data reaches "large" volumes, the system should exercise some optimization strategies to keep the performance at reasonable levels. Larger quantities of data will demand additional resources from a system and although a user will probably realise this, the degradation in performance in the processes and the demand for storage resources should ideally show a linear behavior with repect to data growth. At this point it is important to the user to know the upper limits of a system beyond which the degradation becomes non-linear.

2.2.2 Qualitative Evolution

To answer the second question, two aspects are considered important. First, the ability of the system to meet the needs of users with different levels of expertise and frequency of use. Second, the ability of the system to cope with users demanding an interface that permits them to express their problems in a way that closely matches their mental model.

The first aspect is related to the problems of optimization mentioned before. High-level languages normally facilitate the specification of difficult problems at the expense of relying on the system's ability to efficiently process them. This problem is particularly acute in data independent systems such as those based on the relational model where the freedom gained by the users imposes additional responsibilities on the DBS in terms of optimization, both structural and operational.

The second aspect refers to the advantages of working with a system with which the user feels comfortable i.e. ideally, he perceives that the way the system allows him to pose his problems is adequate. In this sense, it would be desirable to have a choice of ready-made

user interfaces and some means of easily providing new ones e.g. a set of components in the form of "database function specifications" that would permit the construction of a customised interface to a database through function calls from a programming language. Another approach used, e.g. in System R, Ingres and QBE (reviewed later), is to embed a data sublanguage in a host programming language such as PL/1, or Cobol. Still another is to extend a programming language with "database types" such as Pascal/R [Schmidt 77] with data of type relation . Finally, the solution adopted by Atkinson, to extend a language with persistence capabilities, is reviewed in some detail in a further chapter and a combination of it and the "set of components" approach is presented as the chosen alternative in this research and also discussed later.

To cater effectively for all categories of users, a system should be open ended with respect to user interfaces. The objective is to have the ability to provide a tailored user interface in days or weeks rather than months.

2.2.3 Categorisation of Users

To assess the ability of the system to provide suitable interfaces for different users, three categories of users, according to their needs and competence, will be defined.

a) The *casual* user, is a person that infrequently uses the system, and leaves normally more than a week, perhaps months, between uses. This person requires a "friendly" interactive system that is easy to use, forgiving (sympathetic error messages), reassuring (positive feedback) and guiding (takes the initiative and provides context dependent help facilities). It should be easy to remember while using, especially to start using, be self documented (users don't want to carry a manual nor to read it each time they want to use the system).

Casual users are expected to pose, in general, "simple" queries avoiding the use of formal constructs (e.g. logical operators) which they are liable to misuse [Cuff 79, Vandijck 78].

The user interface will typically be a natural language or a non procedural, relational calculus based language. A non procedural language permits the user to "describe" what the solution of a problem should look like, in contrast to procedural systems where the user algorithmically "prescribes" how to reach the solution or how to "navigate" through the database to arrive at the solution.

Casual users are also referred to as *non-experts*, *non-professional* or *naive*. Cuff discusses the needs of casual users in detail [Cuff 79].

b) The *professional* user is a person that frequently uses the system but is not a programmer (and probably does not want to become one to solve his problems). He uses the system as a tool for his main occupation and therefore may require more suitable interfaces than those provided by default by the system. This user will also require at first an interactive user interface that is easy to understand and that will help him to quickly familiarise himself with the system. Such an interface can be the same as the one provided

for the casual users, furthermore it could be enough for a large amount (perhaps all) of the processing needs of this type of user. He may, although, find it inconvenient in that it doesn't allow him to express his queries naturally, or allow him to pose efficiently complex or sophisticated queries, or may not be properly integrated with the other computer based tools he wishes to use.

This user may find that a simple to use interface is not capable of processing adequately (or not at all) some more complex transactions. He may be prepared to spend some time learning another more powerful language, perhaps of a more procedural nature. A transaction is considered here as a user defined data manipulation request to a database (i.e. a query or an update).

c) The *expert* user is a person who makes heavy use of the system and is prepared to spend considerable amounts of time learning how to achieve proficiency with the system. He may want to access the database through a programming language, build or have built a special purpose interface to suit his needs. With respect to this user, the others are also called *non-experts*.

It is further required that users in the first two categories are able to start their applications and administer them without assistance of other persons (unless a non-expert user requires a special purpose interface). In this respect we emphasize the possibility of having "personal" databases.

Obviously, not all users will fit neatly in any of those categories, but they will help in analysing the capabilities of each system for addressing different user needs.

2.2.4 Operational and Structural Requirements

The processing power requirement for a relational system is to have at least one *fully relational* user interface. A system is fully relational if, according to [Codd 79] and [Schmidt 83], it supports:

1. The structural aspects of the relational model: all information in the database is represented as values in tables (relations) and there are no user-visible navigation links between these tables.

2. A data sublanguage at least as powerful as the relational algebra, in whatever syntax without resorting to iterative loops or recursion. The relational algebra operations are project, select, join and the set operations union, difference, intersection and cartesian product.

3. Integrity rules controlling inserts, updates, and deletions (entity integrity and referential integrity).

Other desirable features of a system are:

a) To have the property of closure under the operations i.e. the result of an operation can

be used in another operation. This is especially useful for non-experts since it permits the nesting of simple queries to form a complex query.

b) The loading of a database from an external (non database) file and some form of simple report generator or formatted output facility to neatly produce a data file from the database.

c) Aggregation and group-by functions, such as counting, minimum, maximum and sum.

d) Uniform access to data and meta-data for both querying and updates. Meta-data (the "intention" of the database) is data used to describe data (the "extension" of the database) and is normally contained in database schemas and directories. This enhances the simplicity of a language by using the same syntax in both cases.

Finally, other facilities such as graphics, statistics, report generators, etc. may be also provided. A system offering these facilities conforms to a "database environment" and may provide software specifications that can be used to build further components that are compatible within the environment.

It is not necessarily the case that an easy to use interface has limited processing power and in some cases a user with little knowledge may start using one system language and gradually move toward increasingly complex transactions within the same language.

We assume that recovery and security are an integral part of all systems and are not included in the analysis.

To illustrate the different query syntax of some of the systems, a portion of a "University" database, relating students and the courses they attend is used. The description of the relations is shown in figure 2-1. The relation "Attend" relates, through the attributes "s#" and "c#", to the relation "Student" and "Course".

Figure 2-1: University Database Schema

```
Relation Student : s#, name, address, faculty
Relation Attend  : s#, c#
Relation Course  : c#, title
```

2.3 QBE

QBE (Query-by-Example) is an IBM product, released in September 1978, and featuring a comprehensive relational database system intended for a wide range of users, from casual to experts. It has more than 100 applications and 200 users [Zloof 81]. This figure suggests a number of personal or small-group databases.

The QBE language was designed by Zloof [Zloof 75, Zloof 77] and has been seminal in the research and development of various systems in different application areas. One of these

systems has been designed for office automation, where objects are not only relations but forms, charts, graphs and letters. (OBE: Office Procedures By Example [Zloof 81]) Another area which it has influenced is in pictorial systems (QPE: Query by Pictorial Example [Chang 81]). Finally, an extension of QBE has been considered as a common interface to heterogeneous databases (GQBE: Generalized Query by Example [Jacobs 83]).

The distinguished characteristic of QBE is the handling of tables on a terminal screen giving the user the perception that he can manually manipulate them.

Queries are formed by displaying skeleton tables on a screen which the user fills with examples of possible answers, indicating in that way how the result table (resulting from processing the query) should look.

Once a query has been composed, the user may command the system to process it and display the result table on the screen which can then be scrolled in all directions through cursor positioning keys.

2.3.1 Expressive Power

QBE is fully relational, has group functions and is closed under the operations. It also has loading and output facilities. Access to data and meta-data is achieved through the same syntax.

To illustrate some querying aspects of QBE, consider the database in figure 2-1.

To process the query "list the names and numbers of students enrolled in the faculty of science" would require QBE to display a skeleton table with the column names of the "Student" table. The user would then insert the "constant" element "SCIENCE" under the "faculty" column (a constant element indicates what values should be contained in the corresponding column of the result table) and the command "p." under the "s#" and "name" column, indicating that those columns should appear in the result table. The query would be seen on a screen as in figure 2-2.

Figure 2-2: QBE Query on one Table

Student

s#	name	address	faculty
P.	P.		SCIENCE

The result table is shown in figure 2-3.

The query "list the title of the courses attended by the student Jones" can be formed by inserting identical "example" elements in the linking columns "s#" and "c#" to indicate that the result table should contain information from rows in which column values are identical. The example elements are discriminated from the constant elements by enclosing them in double quotes (QBE underlines them). This is shown in figure 2-4.

Figure 2-3: QBE Result of Query on one Table

s#	name
802425	Jones
803421	Smith
823454	Zloof

Figure 2-4: QBE Query on Three Tables

Student

s#	name	address	faculty
"sss"	Jones		

Attend

s#	c#
"sss"	"ccc"

Course

c#	title
"ccc"	p.

2.3.2 Accommodating User Categories

It is claimed that QBE is a highly non-procedural user-friendly system especially suited for non-experts [Thomas 75, Bontempo 80, Zloof 75]. Thomas and Gould have performed controlled experiments with QBE and conclude that

> *"QBE is easy to learn and use in a relative sense and perhaps in an absolute sense also"* [Thomas 75].

Their results also indicate the suitability of QBE for casual users, since the retesting of subjects after two weeks of the initial training showed *"nearly as correct"* results as they did during the training, and that non-programmers needed less than three hours to make "complicated" queries. Zloof reports that at customer sites users learn to use QBE in about a day [Zloof 81].

Thomas and Gould also conclude that the spectrum of users is completely covered. Their tests show that

> *"the language is behaviorally extendable, i.e. a novice user need only learn a small part of it to write successful queries for simple questions".*

It can also be used embedded in PL/I, thus making it suitable for programmers as well.

Pirotte regards QBE as an attractive language for non-experts but considers that the perception of simplicity is lost in complex queries [Pirotte 78]. Our own experience with

Edquse, a QBE-like language, registers the same phenomenon which also occurs in simple queries involving more tables then a screen can hold at one time.

2.3.3 Operational Environment

The operational environment for QBE requires the PL/I transient library and some extensions to the VM/CMS operating system.

The hardware environment requires an IBM S/370 model 135 and up.

On the other end of the spectrum, QBE supports up to 226 megabytes of data making it an attractive candidate for larger databases [Bontempo 80].

Although no figures for software requirements other than the PL/I library were found, the above considerations suggest that QBE will need considerable computing resources. This would make it unsuitable for small applications on small machines. It is not known whether the software is portable to non-IBM machines.

2.4 SYSTEM R

System R is an experimental relational DBMS supporting an english-like language called SQL also called SEQUEL II (Structured English Query Language), It has been developed at IBM Research Laboratory in San Jose, California, and a prototype was completed in 1979.

The main design objective of System R was

> *"to allow easy definition of databases and database applications without sacrificing the function and performance available in most commercial systems"* [Astrahan 79].

2.4.1 Expressive Power

System R is fully relational, is closed under the operations and has group functions. Meta-data is accessed through special data definition syntax, such as CREATE TABLE *table-name*. It has loading and output facilities.

The basic operation is the "mapping". A mapping returns a collection of values from the key phrase:

```
SELECT  (column(s))
FROM    (table(s))
WHERE   (qualification(s));
```

For example, the query "list the names and numbers of the students enrolled in the faculty of science", used to illustrate QBE would be expressed in SQL as in figure 2-5.

The query "list the title of the courses attended by the student Jones" would be expressed in SQL as a nesting of mappings as in figure 2-6.

Figure 2-5: SQL Query on one Relation

```
SELECT  name, s#
FROM    Student
WHERE   faculty = "SCIENCE";
```

Figure 2-6: SQL Query on Three Relations

```
SELECT  title
FROM    Course
WHERE   c# = IN
   SELECT  c#
   FROM    Attend
   WHERE   s# = IN
      SELECT  s#
      FROM    Student
      WHERE   name = "Jones";
```

This hierarchical nesting of mappings shows that the users needs to know the structure well, to pose the queries. It has also been criticised as confusing [Dieckmann 81].

2.4.2 Accommodating User Categories

As QBE, System R attempts to cover the whole spectrum of user categories from casual to experts.

It can be used both interactively and embedded in a host language. The interactive version of SQL is called UFI (User Friendly Interface) and it uses the same SQL syntax as for the batch operation mode in which SQL statements are embedded in PL/I or Cobol.

Reisner has performed controlled experiments using SQL on experts (programmers) and non-expert subjects. She concludes that non-experts learnt the language "with reasonable proficiency" after 14 hours of instruction. Programmers needed 12 hours and learned the language better. One week after the training, both experts and programmers were still able to use the language efficiently [Reisner 75]. These experiments (as those performed by Thomas and Gould) were of the "pencil-and-paper" type, without actually using the language on a live system.

Kinsley reports that SQL has been tested with several databases of "less than 200 megabytes".

> *"User sites were able to install System R, design and load a database, and use application programs in a matter of days"* [Kinsley 80].

2.4.3 Operational Environment

System R runs on IBM 370 machines (series 138 and up), is offered as an IBM product called SQL/Data System and is available only for DOS/VSE systems. It requires storage on any 370 Direct Access Device e.g. IBM 3330 Disk Drive and uses any 370 compatible terminal [Kinsley 80]. An important feature is the provision of a query optimizer.

Musgrave provides some comments by people, outside IBM, who saw System R in action:

> *"most who saw the system reported that it evidenced massive uses of system resources- both machine cycles and memory".*

He also indicates that SQL requires at least 1M byte of real memory [Musgrave 81].

The above remarks would leave System R unsuitable for small applications. However, no figures have been reported on its use on small volumes of data, probably because it was meant for larger scale databases.

2.5 INGRES

Ingres (Interactive Graphics and Retrieval System) is a relational DBMS developed by the University of California, Berkeley and is currently marketed by Relational Technology, Inc. It has more than 100 users [Stonebraker 80].

The design goals of Ingres were

> *"to provide a high degree of data independence and a non-procedural facility for data definition, retrieval, update, access control and integrity verification"* [McDonald 81].

2.5.1 Expressive Power

Ingres is fully relational and has grouping functions. The access to data and meta-data is not uniform since it uses a different syntax e.g. CREATE *table-name* to create a new relation and APPEND to add a new tuple. Closure is achieved by explicitly using the optional clause [*into resultname*] to create the result of a query as a relation.

It also provides the ability of loading files into relations.

A part of the QUEL syntax for queries is:

```
RANGE of tuple-variable IS relation-name
RETRIEVE (target-list) WHERE qualification
```

The query "list the name and number of the students enrolled in the faculty of science" may be expressed as in figure 2-7.

QUEL has been criticized for always requiring a range variable even in simple cases such as

Figure 2-7: QUEL Query on one Relation

```
RANGE of S IS Student
RETRIEVE (S.name,S.s#)
   WHERE S.faculty = "SCIENCE"
```

above [Dieckmann 81]. However, the following example shows that QUEL is less procedural than SQL and closer to relational calculus.

The query "list the title of the courses attended by the student Jones" is shown in figure

Figure 2-8: QUEL Query on Three Relations

```
RANGE of S is Student
RANGE of A is Attend
RANGE of C is Course
RETRIEVE (C.title)
WHERE   S.s# = A.s# AND
        A.c# = C.c# AND
        S.name    = "Jones"
```

2.5.2 Accommodating User Categories

Ingres provides four interfaces. QUEL is a non-procedural interactive query language intended for non-experts. It is based on the relational calculus. EQUEL is QUEL embedded in the programming language C, allowing tuple at a time retrieval. CUPID is a non-procedural pictorial query language designed for the casual user. It utilizes a light pen and was designed for interactive use. GEO-QUEL provides the capability of presenting geographic data in map form [McDonald 81].

Clearly, Ingres covers the whole range of users with a rich variety of interfaces.

Stonebraker reports on some of the comments made by Ingres users [Stonebraker 80]:

- Ingres is too slow (especially for trivial interactions)
- Ingres is too slow for verly large data bases
- EQUEL is not particularly friendly
- The system is easy to use after a minor amount of training. The "start up" cost is much lower than for other systems.

2.5.3 Operational Environment

Ingres has been implemented on top of the UNIX operating system for PDP 11/40, 11/45 and 11/70 and also VAX/VMS computers. The maximum size of a relation is limited by the UNIX filing system but database sizes are unlimited and the system contains query optimization code.

It is primarily programmed in C [Stonebraker 76], having about 1/2M bytes of code, most of it free of documentation, EQUEL "working set" is about 150K bytes, plus the program size [Stonebraker 80].

2.6 LOGIQUEST III

LogiQuest III is a relational DBMS developed by Software Products International (SPI)

> *"which can be effectively used by non-programming personnel"* [Logiquest 82].

Although it is intended for micro computers, the functionality it offers is similar to those packages offered for larger machines.

2.6.1 Expressive Power

LogiQuest accesses data and meta-data using different syntax. The user is provided with a menu list containing options such as *make file* to create, extend or modify a relation. The make file command is used to specify or change the maximum number of tuples in the relation, add, delete or modify attributes of the relation, etc. The *new* command is used to add a new tuple to a relation or modify a relation.

It has flexible report generating facilities allowing the user to design printout formats. Relations may be also associated with user defined input and output masks for entering and displaying data on the screen. Closure is achieved through the creation of *virtual* files resulting from query evaluations (*find* operations) that may be used in further operations.

The query "list the name and number of the students enrolled in the faculty of science" is shown in figure 2-9.

Figure 2-9: LogiQuest Query on One Relation

```
From:  Student
Select:  name,  s#
Where:  faculty  =  'SCIENCE'
```

LogiQuest has no set operations (although an *append* facility exists), grouping functions (sum, mean, count, max and min) can be defined for report generation but not in queries. There are no integrity constraints and joins may be defined on up to five relations only. It is therefore not relationally complete.

The query "list the title of the courses attended by the student Jones" is shown in figure 2-10.

Figure 2-10: LogiQuest Query on Three Relations

```
From:  Student, Attend, Course
Select:  Course.title
Where:  Student.s# = Attend.s# AND
        Attend.c#  = Course.c# AND
        Student.name    = 'Jones'
```

2.6.2 Accommodating User Categories

LogiQuest can be used interactively through INQUIRY, the SQL like query language used in the previous exmaples and through Pascal programs. The masking facilities can be used to pre-format the layout of the relations and query results on the screen to suit particular applications or individual preferences. Although the interactive facility has relatively limited processing capabilities, more demanding applications may be programmed in Pascal using procedure calls to the available database routines. Logiquest is a well documented menu-oriented system providing adequate help and guiding facilities to casual users. In addition, a user may define a set of instructions, including queries and associate a name with them. Logiquest will execute the entire set of instructions after entering their name. This facility can be used to define complex or repetitive processes to be used by non-expert users.

2.6.3 Operational Environment

LogiQuest III was developed in UCSD Pascal and is currently being converted to CP/M. It runs on IBM Personal Computer (not needing UCSD), IBM Displaywriter, TRS-80 and Altos 8000. It will be soon available for Apple II and III and DEC Professional 325 and 350.

It requires UCSD Pascal version IV, 64K memory, a minimum of 1/2 megabyte floppy disk system and a 24 lines x 80 columns terminal. A relation may have up to 32,000 records and up to 55 attributes.

LogiQuest III is integrated to all software produced by SPI. For example, database files (relations) may be used by the financial modelling and report writing system Logicalc.

2.7 Other Systems

A number of other systems could have been included but either the lack of information on some of its aspects (notably user experience) or some inherent limitation made them unsuitable for comparisons. Some of them which were "on the border" of being included are briefly presented.

2.7.1 ORACLE

Oracle is a relational DBS marketed by Relational Software, Inc. and is intended for *"naive users requiring a flexible DBMS"*. It runs on IBM 360 and 370 machines and also DEC PDP-11 and VAX-11 (all models).

It is not fully relational since neither set operations nor integrity constraints are supported. It has grouping functions and a different syntax to update data and meta-data. The user interfaces are based on SQL, which can also be embedded in a host programming language. It provides a "User Friendly Interface" (UFI) with some text editing capabilities. [Driver 80].

2.7.2 RAPPORT

Rapport is a portable relational DBMS designed by Logica for micro, mini or mainframe computers. A subset of it, "Micro Rapport" runs on Z-80 based microcomputers. The interactive interface is not relationally complete (it has no dedicated join) and the embedded language relies on the power of the host language. The host language can be Fortran, Cobol or Coral.

It has no set operations but, as with joins, they can be expressed procedurally with embedded "Search" loop in which relations can be processed in a record-at-a-time basis. Group functions have been also omitted [Lacroix 81].

2.7.3 FORAL LP

As QBE, FORAL LP [Senko *77*] is a two dimensional database language that uses a light pen to interact with the database. The user is presented with a binary semantic network on the screen on which the user can construct queries by touching the nodes and arcs with the light pen. Each touch is immediately checked for errors. There is no need for a keyboard since the screen display includes the alphabet, digits and a menu of keywords which are "activated" by touching them with the light pen.

Foral presents a database in terms of entities and their relationship. The user is not aware of the existence of relations or tuples.

2.7.4 RISS

Riss is a relational DBMS for small systems, its name stands for Relational Inquiry and Storage System.

RISS provides two interfaces, one for non-experts and the other for programmers by calling upon a set of primitive functions from an applications program. The applications program interface allows retrieval in a tuple-at-a-time form.

Their designers comment that retrieval involving more than one relation is complicated by the fact that a temporal relation has to be created explicitly using joins and all the relations involved in the query [McLeod 75].

RISS runs on a PDP-11 computer using the RSTS-11 operating system.

2.7.5 EDQUSE

Edquse [McDuff 80] is an Edinburgh University research prototype which borrows the two dimensional presentation of queries from QBE. It has been upgraded by [Wilson 81] to include updating capabilities through a simple screen editor.

Edquse has been proved to be easy to use for simple queries i.e. those involving as many tables as the screen can hold at any one time. Beyond that, the management of a query becomes difficult.

It is restricted to small volumes of data and provides limited computational capabilities (only project, select and join operations), has loading and output facilities. Meta-data can only be partially queried and needs a special syntax for updates.

Edquse is a stand alone program and the only alternative for users wanting to process an Edquse database with another language is via reformatting the database for the Astrid system simulator [Bell 80] through a database translator program written by the author. This procedure is only recommended for very small databases.

It is not relationally complete since no set operations have been incorporated. The property of closure is not provided.

Edquse operational environment requires the IMP runtime library and runs under VAX/VMS.

Edquse is therefore a good candidate for the casual or non-expert user but it is unsuitable for complex processing requirements or large quantities of data. Hence it is not evolutionary.

2.7.6 Databases for Micro Computers

A growing number of software packages are being offered as database management systems for micro computers. Although in general they are little more than small-file handlers, some of them are achieving the level of functionality normally reserved to DBS on larger computers, such as LogiQuest described earlier. Two others, Condor III and dBase II are also offered as relational systems.

Condor III is a Condor Computer Corp. product, written in assembly language and running on any machine having CP/M, MP/M, MS/DOS or CP/M-86 operating systems. It requires 56K bytes of memory and the maximum file capacity is 32,767 records (although files with more than 128K bytes cannot be sorted). It has a report generator, but the query language exhibits no closure.

dBase II is an Ashton-Tate product, runs on any machine with the same operating systems as the Condor. It requires 48K bytes of memory (128K for the 16 bit version). The maximum file size is 65,363 records. It has a report generator and a different syntax for accessing the data and meta-data.

2.8 Conclusions

The above presentation shows the variety of approaches taken to suit non-expert users (the BCS report lists 136 different systems [BCS 81]) as well as the significant amount of resources apparently used by the larger systems.

Some of the query languages have been tested in controlled experiments but as yet, no conclusive results have arisen as to which is best. Probably each of the languages has some outstanding features that makes it attractive for a particular group of users on their applications. However, these same users may prefer another language for the same application at a later stage e.g. if the application starts as a small database, then QBE may be a good alternative for familiarising users with the system. Once the application grows and users begin to issue more complex queries, another language e.g. QUEL may be more appropriate. Note that although some learning cost is involved in this process, this includes only the syntax of the new language, the object names used remain the same.

The opposite view i.e. that of having one language for all types of user has been proposed by Vandijck. He recognises different needs and levels of expertise in the users of query languages and suggests a "complete" language in which two levels of operations, a simple and a complex one, are provided [Vandijck 78]. This alternative is attractive in the long run in those applications that will be used by approximately the same persons for a long time i.e. users that eventually become experts. The initial training cost may be high if the language has a procedural flavour but it may be lessened by the users interaction. It can also improve compatibility among applications programs by using the same language on a different database, e.g. similar routine tasks such as loading the database, report generations, etc. can be handled by nearly the same software. On the other hand, the apparent economy gained by having a single piece of software that has to be maintained can be hindered by its large size. Another argument against this approach is that the functional richness of a system can intimidate the more naive users [Demers 81].

The uni-language approach does not cater for a variety of other dedicated uses of a database e.g. sophisticated report generators, statistics packages, graphics packages, etc. Some users may regard the use of functions such as "sum" and "average" all that is needed in a query language for statistics, another group may regard as obvious the provision of direct functions to do regression analysis and plot all kinds of graphs. Still another group may not want to see numbers or strings in a database but pictures or forms. Thus, a "complete" all-in-one language system could become more of a limitation if the database is going to play a major role in information handling.

In a less closed environment where casual, professional and expert users interact with different requirements, the multilanguage approach appears more attractive. The provision of

a database software toolkit upon which a database environment is built and enhanced alleviates many of the maintenance problems and serves as a basis for productive software development. Considering the potential users and uses of a system, the emphasis is put on balancing usability and functionality. This approach is presented in the next and later chapters.

Chapter 3
DATABASE SOFTWARE ENGINEERING

3.1 Introduction

It is common that articles addressing software development problems refer to Software Engineering as a discipline which is still in its infancy [Henderson 81, Hoare 76, Wasserman 81]. The "software crisis" argument is usually associated with both the development of software and its subsequent operational maintenance. The former is typified in reports on poor programmer productivity, overspent project budgets, rescheduling of projects, etc. The latter is appreciated in figures demonstrating its comparatively high cost: 49% of the total software development cost [Boehm 79] (70% [Krakowiak 81]) (80% [Wegner 82]); in the number of computer professionals dedicated to it: 66% in USA [Morrissey 79] and in the scarce attention received in the literature: less than 5% of the software engineering literature [Morrissey 79].

The reasons for having this situation is often attributed to the lack of a generally accepted conceptual background on techniques and methodologies as compared to other engineering disciplines [NBS 81, Henderson 81]. The problems are known but there seems to be disagreement on what are the solutions and on how to apply them [McCracken 82, Gladden 82]. Other Engineering disciplines have matured with time, improving their tools and methodologies through the collective experience of their practitioners, providing today reliable frameworks, accomplished examples and a large amount of learning material for their newcomers.

These problems became more relevant once the advances in computer technology, together with the rapid decrease in hardware prices opened perspectives for wider computer applications. Dearnley notes that

> *"over the fifteen years following 1964, programming productivity increased five-fold, as compared to the hardware price-performance ratio increase of one thousand-fold"* [Dearnley 83].

The software crisis was approached in the early 70's through individual techniques such as structured programming [Dijkstra 72] and modularisation [Parnas 72]. This trend has shifted toward understanding the software development cycle as a whole, integrating techniques and methodologies in a coherent framework. It is no longer sufficient to deal properly with

software at the design and coding stages of a system. The productivity of a programmer team and the quality of their product is being engineered from its beginning, when the requirements are analysed between analysts and customers, to the end, when the product has gone into production and the maintenance costs must be minimised.

It is interesting to observe the increasing importance assigned to the more 'human' activities involved in producing software, thus, overlapping with disciplines like management [Brooks 79] and psychology [Weinberg 71, Anderson 80].

Although many solutions have been proposed, there is still little experimental evidence on their effectiveness. Among the areas most frequently reported as needing more research are reusability of components, fast prototyping and software standards [NBS 81]. This chapter addresses these topics first in a general perspective and then in their relation to database software engineering.

3.1.1 Standards

This is probably one of the most controversial issues in today's software engineering debates. It is argued that it is premature to set standards in a still unstable and fast developing discipline. It is, however, recognised that the provision of standards can have a stabilising effect, providing an agreed basis for further developments, diminishing production costs by allowing the reuse of proven technology and the construction of portable systems, reducing technology transfer costs, reducing training costs and widening the usability of systems built around standards by encouraging manufacturers to produce them [NBS 81].

The growing availability of personal computers in the hands of people who are not (and do not wish to become) professional software developers is opening a market for complex but inexpensive software. Software companies may minimise the costs of producing this software by applying techniques of mass production such as standardization of (software) components [Wasserman 82]. However, the standardization of these components will only be useful if they are restricted to the specification of their interfaces and not to the specification of *how* they should work.

Perhaps the counterarguments outweigh the advantages depending on the level where standards are proposed (or imposed). If standards are set up too close to the man-machine interfaces, they may become too restricting for users having a different appreciation of what these interfaces should look like. At the other extreme, if standards are set too close to a specific machine configuration, they will undermine the possibilities of adapting to advanced hardware technology.

3.1.2 Software Components

Managing complexity is perhaps one of the most difficult intellectual endeavours. This is also true in programming, particularly in large programs and it is by means of the powers of abstraction that we may overcome our limitations to deal with complexity [Dijkstra 72]. The

complexity of computer applications will continue to grow rapidly (robotics, language processing, etc) as cheaper and more powerful machines become available. Assembling (or understanding) a complex system as a hierarchy of subsystems ("divide and conquer") is a standard practice among other engineering disciplines.

Rigorously tested software components, with well defined interfaces and adequate documentation on their functionality should make system building easier, better and cheaper than it is today [Belady 81].

Components and their associated documentation should be designed with their intended users in mind. Interface specifications can be provided both informally, particularly for end users, and formally.

> *"The specification must provide to the intended user all the information that he will need to use the program correctly and nothing more"* [Parnas 72].

Confusing or irrelevant technical jargon in documentation intended for end users can only be intimidating. For those implementing, maintaining, verifying or testing a system, in addition to formal and informal interface specifications, a description of the component's internal behavior will be required. Hence, both users and documentation must be categorised [Wasserman 81].

To be usable (reusable), components must be flexible so as not to impose unnecessary restrictions on applications, to cripple the user's creativity or fail to adapt to technological advances. They shouldn't force users to alter their way of doing or thinking about their work [NBS 81].

To achieve the required flexibility and interchangability, components should not be overly large, limiting their functionality to a very narrow scope. They should not be engineered into final "tools" but rather into small components out of which users can build a tool tailored precisely to their needs [NBS 81]. This will make systems more maintainable by facilitating the location of errors and inefficiencies in small components which are then either corrected or replaced. Components themselves will be more maintainable if their interaction with the rest of the system is kept to a minimum [Krakowiak 81].

A component should be robust: it should, for example, survive user abuse [NBS 81].

Despite its inability to communicate structural representations among components, the possibility of composing programs out of components is usually regarded as the key for the success of the UNIX system.

However, a potential disadvantage of components is pointed out in [NBS 81]:

> *"If the tool fragments are too general and reconfigurable, then they cannot exploit any significant knowledge of each other's working, only each other's interfaces.*

Perhaps the solution is to arrive at an adequate compromise, with different levels of fragmentation and intercomponent coupling. This, however, requires a more elaborate organisation of the components and their documentation to make them accessible. The

availability of a component library which includes documentation and perhaps examples will have a major impact on the usability of the components.

3.1.3 Prototypes

A prototype system is a partially complete functional model of a target system. Prototyping relies heavily on the concept of reusable components. One of its major advantages is that a prototype permits both final users and designers of a system to observe its functional behavior early in its development cycle. This may lead to significant improvements in software production and quality by correcting misunderstandings at the specification and design stages when it is much cheaper to do than after the system has gone into production. Boehm notes that

> *"IBM's Santa Teresa software organization has found, on a sample of roughly 1,000,000 instructions of software produced per year to IBM-determined requirements, that the average project experiences a 25% change in requirements during the period of its development"* [Boehm 79].

It is often difficult for the users to visualize how the final system will actually perform just by reading a paper specification. This problem is particularly acute if the users have had no experience with computers or are afraid of changes in their working habits or are even suspicious of the utility of the new system. In this sense prototypes can be an invaluable help as a training tool and to observe and correct user resistance. For example, it is possible to experiment with the user interface from a human factors perspective, particularly if the prototype is for an interactive system, so that it can be tailored towards user preferences [Wasserman 82]. If the final users are allowed to have an active role during the production stages of a system, by permitting them to comment and have an influence on its outcome, they will probably feel more willing to accept it [Dearnley 83]. A prototype may help users understand their own requirements, and iterate over them. Hence, a prototype, like "a picture is worth a thousand words", may help bridge the communication gap between analysts and customers [NBS 81].

A prototype may highlight in advance unforseen technical problems in the system implementation when there is still time to modify specifications. It may also provide invaluable experience on how to implement the final system. Since building a prototype may be regarded as a small project itself ("a poor man's design") possible strategies to be used in the final system can be tried during its development.

Prototypes have, nevertheless, potential disadvantages. They may be regarded as an apparent waste of time by the customers who don't want to see the work duplicated and argue that building a prototype will only delay the production of the definitive system. Some of the potential users may have been against the production of an automated system in first place and may attack the whole project based on the weakness or incompleteness of a prototype. It is also argued that prototyping could be expensive. But Gomaa and Scott report that only 10% of the total software development cost was due to prototyping in their experience of developing a system to manage and control a semiconductor processing facility. They conclude that their prototype proved an invaluable help in revealing requirements and design problems early [Gomaa 81].

The objective is not to build disposable code but to use it (or parts of it) in the definitive system. The prototype becomes the first version of the system which is analyzed and transformed into a subsequent version in a sequence leading to the final system. In that sense, the provision of an adequate pool of software components may greatly reduce the potential disadvantages of prototyping. Other disciplines use prototypes such as pilot plants, cars and aircrafts for the same reasons given above.

As for research activities, where user, designer and implementor is often the same person, fast prototyping can be helpful in at least two ways. First, as an aid in clarifying ideas by facilitating the quick production of a working example which can be observed and second, a prototype may be sufficient to demonstrate that a proposed idea is feasible. It is important that the researcher's software development environment has the necessary tools and components so that the elaboration of experiments is effectively supported.

Prototypes, if correctly scaled in complexity and cost in relation to the final system may help creating cheaper, successful systems in less time.

3.2 Database Software Engineering

Users of data are no longer just experienced programmers. Database technology is being made accessible to an increasing number of persons with different backgrounds and diverse data processing requirements. These users are putting a large variety of sometimes contradictory demands to database applications. There are those who want simple to use, friendly packages that allow them to specify what they want leaving to the system to decide how to do it. Others may be prepared to sacrifice simplicity to achieve more computational power. Still others may just want another way of modeling and processing their data. Finally there will always be those who will need optimum performance and will want to know all possible details they could take advantage of, to instruct the system how to do a job better.

The increasing number of inexperienced users and different application areas, together with the need to improve programming productivity and to protect applications from environmental changes is pressing toward higher-level, data independent programming environments. Morrissey notes that

> *"The widespread adoption of high-level programming languages have made possible productivity improvements of a factor of 3 to 5 in the development and maintenance of software"* [Morrissey 79].

This situation clearly suggests that no data model and no query language will suit everyone. What is needed is not another all purpose database package a la PL/1 but an environment which allows match to the user's need through the composition of an applications package out of existing components.

In what follows, standards, components and prototypes are reviewed in the light of database software engineering. A number of proposals are put forward regarding possible structural and operational standards.

3.2.1 Standards in DBMS

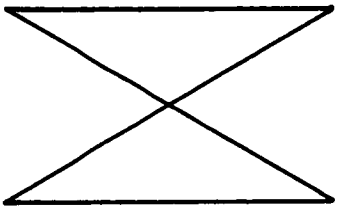
Having recognised the variety of potential users and accepted the need to support their way of performing a job it is easy to realise that at the other end of the spectrum, the hardware, the situation is similar. Hardware technology is one of the fastest changing disciplines, hence, any application which intends to run in the computers of the (near) future cannot be tightly coupled to any particular hardware configuration. As a corollary, any attempt to standardise at the user or hardware level of the computing spectrum is bound to be shortlived.

But users and hardware are the opposite ends of the spectrum. A multitude of approaches at these ends should not only be permitted but also encouraged. However, continuing with such a variety at intermediate levels of the software spectrum is not only unnecessary but can also be wasteful. The objective then, is to find a few intermediate tasks which are general enough to effectively support a large number of user approaches and with reasonable machine independence to facilitate their adaptation to new hardware.

The major advantage of having internal standards, with a set of software components implementing and supporting them, is that further software development costs may be reduced by avoiding the implementation of a great deal of nearly similar software.

This may have an important stabilising effect if users come to rely on those components and use them as a basis for their applications. Software built around the standards may become portable to other sites using the same standards. A variety of components may initially grow around these convergence points. It is expected, however, that this growth will stabilise with time and the efforts in software production will be devoted primarily to suit specific applications and to accommodate the software to new equipment.

A database environment, where its software components are implemented around a few internal standards may be visualized, in its simplest form, as a double cone:



Scope of Standards.

It is recognised that it is not possible to propose standards to suit all imaginable present and future software applications or hardware advancements. For example, standards being general purpose by nature, cannot address effectively the problem of critical efficiency. What is important is to identify an area which is sufficiently large and mature to accommodate and obviously benefit from them.

For these reasons the line drawn in the following discussion encompasses relatively small

applications, intended for interactive database applications using primarily high-level languages. The emphasis is on functionality and modeling capabilities rather than efficiency. Although the latter is particularly important for interactive applications, it is believed that, considering the above constraints, reasonable performance can be achieved. The "smallness" is obviously relative to the available technology, but it is intended that the propositions made can be effectively implemented in personal computers. As the Introduction chapter pointed out, our goal is to extend the facilities of database technology to a larger number and variety of users.

Requirements for Standards.

As a result of the widespread acceptance of the relational model and its advantages in terms of data independence and high-level operations, it has already been considered as a good candidate for a standard data base model:

> *"The purpose of DBMS standard is to facilitate the development, evolution and portability of database applications together with the associated personnel and data, in the face of inevitable and necessary change. It has been claimed, and in many cases demonstrated, that the relational database model contributes substantially to the fulfilment of those goals"* [Brodie 80].

On the other hand, the increasing amount of research in the area of "semantic-oriented" data models suggests the convenience of incorporating in the standards more abstraction mechanisms and higher-level operations than those in the Relational Model. This would not only significantly widen the scope of application for the standards, but also improve their life span.

However, this last consideration has two potential disadvantages. First, such an ambitious scope may increase the complexity of the software. This may directly contradict the requirements for small, flexible components set before. Considering that the "semantic-oriented" candidates are less well established and that few implementations have been reported, the complexity introduced by their incorporation may restrict the freedom required for experimenting with the components. Second, the issue of efficiency becomes more important since the translation of high-level processing requirements over possible highly-structured objects into canonical forms may become intolerably inefficient.

Candidates subject to standardisation.

a) Structure: The ANSI/SPARC committee [Tsichritzis 78] has already proposed a three-layer schemata for database applications:

1. The *External Schema* or data as seen by the users. It defines the application view of the database. All external name resolution is done via the external schema.

2. The *Internal Schema* or data as seen by the system. It relates to the performance strategies employed by the DBMS, the storage structure of the data and access paths (e.g. inverted files, indices, etc.).

3. *Conceptual Schema* which is the enterprise's description of the information as modelled in the database. It contains the definition of entities, their properties, relationships and constraints.

The internal and external schema must be consistent with and mappable to the conceptual schema.

Only one of them, the external schema is given the possibility to replicate itself to accommodate different user views of the conceptual schema and although one conceptual schema is assumed to exist, there seems to be no agreement on what form this schema should have or what should go in it. Perhaps such disagreement comes from the fact that the conceptual schema is still too close to human appreciation and should therefore also be allowed various forms, based on an internal conceptual schema (ICS) which supports this variation [Stocker 81].

Stocker has proposed such an internal conceptual schema:

> *"It is proposed here that the structure of the conceptual schema should be resolved further so that it is interpreted as two views connected by a mapping. One view, the Outside Looking In (OLI), is apparent to users who need to reference the conceptual schema directly. The other, the Inside Looking Out (ILO), is that seen by other system components which reference the conceptual schema. It is suggested that the natures of these two views are quite distinct and that it is the ILO which may more fruitfully be subjected to standardisation studies"* [Stocker 81].

Codd when proposing an extended relational model (RM/T) expressed:

> *"Meaningful units of information larger than the individual n-ary relation have been introduced in such a way that apparently competing semantic approaches recorded elsewhere may all be represented therein and translated thereto"* [Codd 79].

The ICS, should consist of a set of relations. In this form both data and meta-data may be manipulated through the same language. This uniformity has an important effect in the simplicity of the data management system. The relations in the ICS contain information about other relations (e.g. the tuples in the "attributes" relation contains information of all attributes of all relations in the database). These meta-data relations may also support data abstraction (e.g generalisation). Any update on these relations will trigger a number of operations. For example, inserting a tuple in the "attributes" relation requires the addition of an attribute in some table, and the possible extension of existing tuples in that table by applying default values.

A further chapter explains the decision of using a subset of the RM/T as an internal conceptual schema in our research. Basically, the decision was taken not only because of the structuring capabilities of RM/T but also because it contains a well defined set of operations and integrity-preserving rules associated with the structures. Mapping simple queries posed in the functional language Daplex into a relational language (QUEL) has been shown in [Shipman 81] and investigated in the implementation of Adaplex [Chan 81]. It will be shown in chapter 4 that the RM/T is capable of supporting the abstraction mechanisms of Daplex such as subtypes.

b) Operations.

> *"Structure without corresponding operators or inferencing techniques is rather like anatomy without physiology"* [Codd 79].

Two internal languages can be identified as candidates for standardization. The first, which is necessarily a complement of the ICS, is an Internal Data Manipulation Language (IDML). This language will typically create, delete and modify objects in the database (e.g. add tuples, delete tuples). The other language is an Internal Query Language (IQL). Applications can map their queries into the IQL which are subsequently processed by a common evaluator component which implements the operations in the IQL.

We have investigated a subset of the operations contained in the RM/T, as a candidate for the IQL. This language may take the form of a relational algebra where user identified objects e.g. relations, are translated into ICS identifiers. Although the higher procedurality of relational algebra compared to relational calculus should make it easier to process an algebraic query, there is no theoretical reason for not choosing a calculus based language instead of an algebra [Codd 71]. The decision of using an algebra relates more to the fact that we have experimented already with the mappings from a calculus language to a relational algebra using EDQUSE [Wilson 81] and ASTRID [Bell 80].

3.2.2 Components in DBMS.

Considering only the ICS, IDML and IQL as internal standards already suggests a variety of software components around them. The reasons for choosing these components and their description is detailed in chapter 6 and are briefly outlined here.

The Internal Query Language already suggests the existence of a first component: a single abstract query evaluating machine that processes all requests mapped into the IQL. A standard IQL and its associated common evaluator machine have a variety of benefits such as the localisation in one component of all the query processing methods. To improve performance, this component may have some knowledge of the structures manipulated by the logical storage handler. A Logical Storage Handler will provide independence from physical data manipulation. This component implements the IDML, providing a set of functions to create, delete and manipulate objects. The Names and Message Handler is another important component. It translates requests expressed in terms of the external schemata into requests expressed in terms of ICS identifiers. The Updates Handler is in charge of enforcing integrity rules during updates. It may also have to trigger the necessary operations to ensure the consistency of the database. The Query Optimiser could become a very important component if performance is a critical factor. In fact two optimisers could be considered, one at the logical level where no knowledge of data structures is on hand, and a physical optimiser that is tightly coupled to the storage handler and knows about data structures and access paths e.g. it may use information about indices. The View Handler and Version Handler may be included to provide different perceptions of the database and to allow experiments (versions).

A list not much larger than the above should give enough room for a variety of applications.

The main task of the end-user components is to map their requests into the IQL. Some criteria to judge the merits of the IQL and its associated common evaluator as internal standards are related to the following questions:

1. what is the complexity in the mapping of user requests into IQL ?

2. what is the difficulty in implementing the common evaluator ?

3. what is the efficiency of the evaluator ?

4. what is the independence of the evaluator from other components ?

To answer these questions we implemented a number of user interfaces using those components and standards (see chapter 7).

A database environment implemented around the standards offers the following advantages:

1. It allows different query languages and other end-user packages to use a common data base, providing therefore several alternatives from which a user may choose. The choice may depend on the user's experience or on the suitability of the language for his task.

2. It localises implementation efforts at the physical level in a single evaluating module per physical organisation of data, thus alleviating the need for a great deal of near duplicate software.

3. It makes it possible for one front end to refer to many different data bases with possibly different implementations. Query languages which have been mapped into the canonical form may operate on any data base which supports that canonical form and responds to the standard IQL. Hence, languages at the user level become non system specific.

3.2.3 Prototypes in DBMS.

The approach of incremental development of systems should have a positive impact on DBS. Many systems have been reported that have undergone several revisions and rewriting of the software. (Ingres has approximately 1/2 Megabyte of code and has been rewritten entirely several times) [Stonebraker 80].

The prototyping activity should be facilitated by the previous considerations. The prototype components are naturally included in the final product which is implemented around the same components and standards. It has been suggested that any database project, independent of its initial goals, will only have an acceptably low level of risk if it is developed incrementally [Gradwell 81].

Once the mapping from logical to physical storage handling has been engineered for a particular environment, it is only the user end which remains open. This level may be further narrowed, if desired, by implementing other higher-level components such as I/O packages, view handlers, etc which are connected at their lower end with the rest of the components. For this research, the physical administration of the objects in the database was provided by the persistent facilities of the language PS-algol (see chapter 5). Hence, the lowest layer of data handling is the Logical Storage Handler component. A previous version used Pascal and the CMS [Atkinson 83a] interface as the lowest layer. At the upper end, a simple screen manager was used for one of the query languages. It may be expanded into a full multiple-window manager with graphics capabilities that could also be standardized.

3.3 Conclusions

Two issues related to software engineering are important: one is the identification and building of re-usable (possibly standard) components and the other is the development of re-usable experience i.e. techniques and theories that can be reapplied. This research is an attempt to tackle both areas by experimenting with the ways in which DBMS can be partitioned and constructed. The desirability of it comes from the need to build new databases, the difficulty people presently experience doing that and the changing need for single or small-group databases with small quantities of data.

The identification of software standards that can support a variety of end-user applications and adapt to a varying hardware technology have great potential in reducing software development and maintenance costs. A database applications environment may be initially set up around several software components implementing the standards. Further software development concentrates at levels beyond the standards. Prototyping, considered as a fundamental tool for reducing software development and experimenting, is significantly facilitated by this approach.

A subset of the extended relational model [Codd 79] appears as a good candidate for internal standards. It is investigated in more detail in chapter 4. A large variety of existing query languages, based on possibly different data base models may be supported by this approach.

The problems of scope of standard and efficiency remain as open questions and it is expected that the present implementation of a database environment for small, interactive applications will show where to set the upper limits of complexity and volumes of data that can be reasonably handled.

Chapter 4
DATA MODELS

4.1 Introduction

The convenience of having internal database standards in the form of an *Internal Conceptual Schema* (ICS) and its associated *Internal Data Manipulation Language* (IDML) and Internal Query Language (IQL) was discussed generally in chapter 3. The ICS, IQL and IDML constitute an *internal* data model which is expected to be able to support a variety of *external* views of the ICS [Stocker 81]. In this chapter we elaborate on those propositions and discuss a data model that is well equipped to play the internal role upon which other data models may coexist. In fact, only the external and conceptual schemas of the other models coexist since they are based on the common ICS and it in turn, is mapped onto a unique internal schema (this is assumed in the rest of the chapter when referring to the *other* data models.

We are interested in what should be the constituents of a "good" internal conceptual schema. We also want that schema to be amenable to computer processing. The methodologies for, and the process of analysing a given reality leading to a conceptual schema is beyond the aims of this research.

The next section discusses some issues in data modeling which have an impact on the concepts needed in an ICS and their relevance to existing data models. Further sections present the chosen internal data model and the mappings from external schemas to the common ICS.

4.2 Why Coexisting Data Models ?

Nijssen refers to the portion of the reality which one or more persons agree to select as their common frame of reference as the *Universe of Discourse* (UoD). The description of the classes of objects of the UoD, the relationships among those classes and their constraints is called a *Conceptual Schema* (a constraint is a property of an object which is either true or false). Different groups of people may want to view only a portion of the UoD and in a way which is different from the inherent mental model in the common UoD [Stocker 81]. This view may be a subset or a *transformed* subset of the UoD [Nijssen 76].

The importance of viewing the UoD under different mental models is reflected in the lack of agreement on a generally accepted "super" model, in the many models proposed so far (twenty-three are discussed in [Kerschberg 76]) and in the results of recent analysis on user behavior (see chapter 9) which suggest that a person will perform best if operating under his preferred mental model.

Nijssen is of the opinion that the next generation of generalized data management systems has to provide the user with the option to select his preferred mental model and his preferred manipulation language. He refers to this approach as the *coexistence* model [Nijssen 76].

4.3 Requirements For The Coexistence Conceptual Schema

In searching for an appropriate internal conceptual schema as a basis for a multi-model, multi-language database system, we are interested in those data models which offer not only a collection of structured data types but also include operations over the types (evolution rules) as well as integrity constraints. We will attempt to identify those concepts useful for a data independent framework in which the definition of the information content of an application can be stated as precisely and concisely as possible so that it can be effectively used as a foundation for developing coexisting data models.

Many suggestions with respect to requirements for a "right" model can be found in the literature. We list some of them which are relevant to an internal data model. A data model should:

Allow a concise and understandable representation of a system at any point in time. Allow high-level access to the information. Have a set-theoretic personality to allow search without navigation. Have a generalised graph structure, suitable for graph-theoretic analysis and for a more attractive presentation of objects in the database to certain classes of users. Have entities and relationships as primitive concepts. Allow entities to belong to many categories (generalization and aggregation abstractions are discussed later). Not confuse entities with their names and representations. Not display anything readily recognizable as a record [Kent 76, Codd 79, Smith 77].

With respect to data independence, it is desirable that the ICS has no reference to physical constructs which should be contained in the internal (storage) schema, so that programs benefit from data independence and do not need to be modified if the database is physically restructured. Kent considers that:

> *"the conceptual schema should reflect information rather than data processing technology"* [Kent 76]

4.3.1 Records versus Entities

The record has been a traditional modeling primitive in database applications but some of its disadvantages have prompted the alternative concept of entities as more appropriate. An entity has been loosely defined as a "thing" that exists and is distinguishable [Ullman 82, Date 83].

Kent argues against the record oriented approach reflected in the traditional data models i.e. hierarchical, network and relational (for tutorials on these models see [Date 81, Ullman 82, Tsichritzis 82]). He refers to record oriented technology as

> *"an ingrained habit of thought coming from the early days of data processing where real business was record processing almost by definition"*

he continues:

> *"the record oriented tradition imposes biases into the conceptual model which have nothing to do with the information content of the database. The biases reflect the preferred practices, limited capabilities and performance constraints of today's technology* [Kent 76]"

The arguments against record technology concentrate on two observations: First, the asymmetric representation of relations where m:n relationships (e.g. between students and courses) are regarded as independent entities by introducing a third record type whereas 1:n relationships (e.g. between tutor and student) are not treated as independent entities because there is no implementational motivation for it (the tutor identifier is filled into the student record). Second, in the naming of a relationship, where a name is normally given only if the relationship is m:n because of its implementation in a distinct record type that needs to be named [Kent 76].

4.3.2 Surrogates versus User-defined Keys

Surrogates have been proposed as representatives of objects (entities) in the Universe of Discourse. A surrogate is a unique system defined entity identifier. The notion of surrogate has been investigated by [Hall 76, Codd 79] and others to overcome some of the problems of the relational model in uniquely identifying entities through user-defined keys (a key is any attribute or set of attributes that uniquely determines a tuple). Three different problems may arise with user-defined keys:

1) The value of a key may have to be changed. Consider an archeological database where the identification of a bone is by its scientific name. Occasionally bones which were thought to belong to some species are later found to belong to another one.

2) Different keys may be identifying the same entity. Consider two archeological databases which are merged and one identifies the bones through some serial number and the other through their scientific names.

3) Keys may not be applicable (or cease to be). Consider a recently found bone which is still unclassified or a bone that is dismissed as a forgery.

Surrogates have been introduced to alleviate these problems. Each entity is permanently assigned its own unique surrogate which acts as the representative of the entity in the outside world. Users can still define keys but they are not compelled to do so.

It has been suggested that users should have no access (never see) the surrogates themselves but should be able to use them in operations (such as joins). The argument for not seeing them is that since they are generated by the system and are not updatable, their values have no meaning for the user [Codd 79]. Another problem of allowing users to see surrogates is that these are controlled by the database system which may modify them behind the scenes, thus invalidating any entity reference that users may have created using surrogate values explicitly.

4.3.3 Domains

There are two uses normally associated with the concept of domain. First, it is used to define the set of values that are acceptable for a given attribute and second, to indicate when a relationship might be inferred on the basis of common domains. In the second usage, the domain may be viewed as an entity type regardless of how the entities are named or represented.

> *"The domain concept shows a spectrum of sophistication moving from data items to the concept of an abstract thing which might be represented by any member of an equivalence class of data items. The tendency has been to limit such classes to things which can be numerically computed from each other (e.g. date types, units of measure). As an example of domains that are normally not provided in the same equivalence class i.e. without the ability to derive one from another, is the representation of people by social security number in one place and employee number in another place"* [Kent 76].

> *"For the conceptual model, we should have a concept of domain which makes better separation between entities or values and their representation in data items"* [Kent 76].

4.3.4 The Relational Model

In the early stages of this research, the objective was to support a multi-language external world only, not a multi-model one. This external world would consist of both relational calculus and relational algebra based systems (for a tutorial on relational algebra and calculus see [Ullman 82]). Therefore, the relational model as proposed in [Codd 70, Codd 71] was naturally going to serve as the internal data model. This view was stimulated by the availability of two database systems in the department which could be used as the basis for experimentation. One of these systems, called EDQUSE [Wilson 81] is calculus oriented and similar to QBE [Zloof 75]. The other system, called ASTRID [Bell 80] is based on the relational algebra [Codd 71].

Both systems were modified to access the same database by making their querying components parse each query into a canonical intermediate language that was afterwards processed by an interpreter and a common query evaluator. Each system was provided with an external schema produced from a common ICS.

Although the process went through several separate stages and the database was not directly updatable, the exercise laid the foundations for a canonical intermediate language for relational systems. This language has been further extended to support a richer class of queries and is presented in chapter 6.

Since the external world was meant to be relational, an ICS based on relations would not differ significantly from external schemas. The power of the system would rely on the ability of the internal model to support complex processing requests from relational systems, possibly embedded in a conventional programming language. Instead of pursuing this line, it was decided to amplify the users choice of external views by supporting interactive systems other than the relational. The first step in this direction is presented in the next section.

Most of the examples in this thesis will be drawn from a hypothetical *University Database*. The Bachman diagram is shown in figure 4–1. Single headed arrows denote single–valued relationships, double headed arrows denote multi–valued relationships. The database can be described in the following way: a teacher organises a set (of none, one or more) courses and a course is organised by a set of teachers. A course is attended by a set of students which in turn may attend a set of courses. A teacher works for at most one department but a department may have a set of teachers working in it. A student belongs to at most one department and a department may have a set of students.

The description of each relation is shown in figure 4–2.

Figure 4–1: University Database. Bachman Diagram

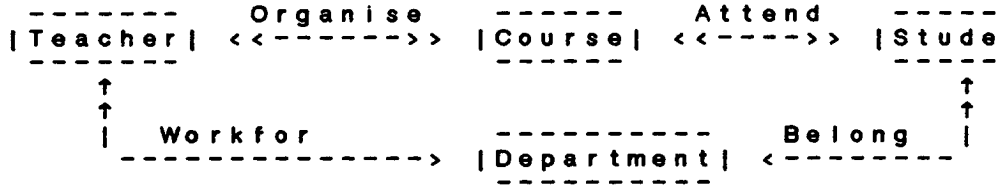


Figure 4–2: University Database. Relational Description

```
Teacher    : t#:integer,name:string,salary:integer
                    phone:integer,ext:integer
Course     : c#:integer,title:string
Student    : s#:integer,name:string,faculty:string
Department : d#:integer,dname:string,head:string
Organise   : t#:integer,c#:integer
Attend     : c#:integer,s#:integer
Workfor    : t#:integer,d#:integer
Belong     : d#:integer,s#:integer
```

4.3.5 The Functional Query Language FQL

After experimenting with the relational systems, it was decided to investigate the broadening of the external world spectrum of data model possibilities. One of the systems proposed as an intermediate query language, based on a functional data model, is FQL [Buneman 82]. We give a brief introduction to it here. Further details of it are given in chapter 7.

FQL is an applicative language, based upon a functional programming system as suggested in [Backus 78], with powerful computational capabilities including arithmetic, relational and boolean operators as well as sequence operators, achieving the same computational power as pure Lisp. Functions can be defined recursively.

Its control structures can be used to support interfaces to higher-level querying systems. It has no notion of data currency and queries (functions) can be developed step by step.

FQL can be used as a general purpose programming language: there is no need to refer to any database while using it.

One of the problems with present applicative systems is the way they represent a sequence of objects. This is normally done using arrays or linked lists. For database applications, where a sequence of objects can be very large, this representation cannot be used. FQL overcomes this problem by representing a sequence of objects as a *stream* which is a two component structure whose first component is the first member of the sequence and whose second component is a *suspension*. A suspension consists of a function f and an argument a. The application of f over a creates another stream, the *tail* of the original stream. Suspensions must be evaluated repeatedly to traverse the stream. Using this mechanism, the amount of primary store required to process a sequence of arbitrary length remains constant.

The evaluation of an FQL expression is done in two steps. First, a set of translation rules transform any FQL expression into an internal data structure. Second, another set of rules is applied to evaluate the structure, producing printable objects i.e. strings, integers, etc.

We implemented a subset of FQL, first in Pascal to investigate its suitability as an intermediate language for relational systems, and later in PS-algol as a functional query language.

The Pascal experiment demonstrated that it was relatively easy to produce an FQL schema in which relations define data types and "constant" functions and attributes define functions.

Consider the relation "Student" and its string attributes "name" and "faculty". The corresponding FQL schema would define the following functions:

```
function          function's type

!Student  :              ->  *Student
name      :  Student     ->  String
faculty   :  Student     ->  String
```

The characters "!" and "*" are part of the FQL syntax which is described in more detail in

chapter 7. "Student" is a "constant" function that yields a stream of objects ("*" denotes a stream) of type "Student". "name" and "faculty" are functions that, given an object of type "Student" yield an object of type "String".

However, it was found that it was not clear how to produce an FQL schema that included "link" functions. Consider for example two further relations named "Course" and "Attend". The latter relates students with the courses they attend, that is, "Attend" would define a function that, given an object of type "Student" yields a stream of objects of type "Course".

These "link" functions could be defined by identifying common attribute domains among relations (e.g. "Student.number" in both "Student" and "Attend" relations and "Course.number" in both "Attend" and "Course" relations. This method would overload the concept of domain, as discussed previously and may also produce some unwanted functions due to the proliferation of common domains.

An alternative method is to use referential integrity constraints which explicitly establish inter-relation relationships. E.g. the relation "Attends", whose two attributes are "Student.number" and "Course.number", could be defined with the following referential integrity constraints: *"Student.number" exists in relation "Student"* and *"Course.number" exists in relation "Course"*.

However, the main problem found with FQL was its lack of updating capabilities. Since other data models appeared more attractive in this sense, the FQL experiment was temporarily stopped and later resumed with a PS-algol version of it. This is presented in chapter 7.

4.4 The Extended Relational Model

After analysing and experimenting with some of the ideas presented so far, we came across an extension to the relational model, called RM/T (T is for Tasmania where it was first presented), that complies with the requirements set for an ICS. In addition, it is based on a well established and proven formalism, the original relational model [Codd 70, Codd 71], upon which an increasing proportion of applications are being designed and implemented [Codd 82]. Finally, and since the supporting of relational systems is one of the main goals of this research, it seemed natural to experiment with an extension of it as the internal model.

The extended relational model was proposed in [Codd 79] and later refined in [Date 83] as a means of overcoming some of the disadvantages of the original relational model (henceforth referred to as the relational model) with respect to *semantic data modeling* (this term is used to denote the task of capturing more of the meaning of data). The relational model has been criticised as a "flat" model incapable of representing semantic abstractions such as generalization (sub-types) [Smith 77] and for not having an "object" orientation. Schmid made the following two remarks:

> *"there is often difficulty in understanding how a collection of relations is actually modeling the underlying system" ... "it is difficult to formulate general rules to*

> *guarantee the consistency of the collection of relations as they are updated"* [Schmid 75].

Codd has replied with a number of extensions to the relational model, all of them named RM/T. In RM/T, as in other discussions on data models (e.g. see [Codd 80]), Codd has always stressed the need to incorporate manipulative as well as structural aspects in data models. He suggests that the former has been too often neglected, leading to incomplete models or to meaningless remarks when comparing models. He also indicates that the task of "semantic" modeling is a never ending one and that the label "semantic" is by no means absolute. Finally, he has suggested that the RM/T is capable of representing the structures and operations of other data models. This last fact was one of the main reasons for choosing a subset of the RM/T as the internal data model.

Although the discussion of the whole of RM/T is beyond the scope of this research, a summary of it (extracted from [Codd 79] and [Date 83]), together with the implementational aspects related to this research is presented in the next section.

The fundamental idea in RM/T is that applications can be modeled as a collection of entity types and the relationships among them. A number of structures, called *Graph relations*, are used to represent explicitly the different entity types and their relationships. New operators deal with these Graph relations and a set of rules is defined, in each case, to preserve data integrity. RM/T is also based on the notion of surrogates, as discussed earlier.

The topics in RM/T not covered in the following analysis are the use of null values and the set of operators and rules defined to deal with them. Codd himself regarded this area as "preliminary and in need of further research". Date proposes default values as an alternative to null values. This simplifies the operations because a default value is treated as a "normal" value (instead of the three-state logic approach used with null values in Codd's proposition). Other topics not mentioned are some of the Graph relations used to define abstractions such as Cover aggregation, Conditional generalization, etc. and some other new operations and integrity rules. The objective was to identify the portions of RM/T that would prove its ability as a foundation for the coexistence of external data models.

4.4.1 Entity Types and Immediate Properties

A database may have entities of several types and an entity may participate in different types at the same time. For each entity there is a unary relation, called *E-relation*, that contains all the surrogates of entities existing in the database that have that type. The primary purpose of the E-relation is to assert the existence of entities of that type. The sole attribute of an E-relation is called *E-attribute* and its values are taken from the domain of all entity surrogates called the *E-domain*.

An *immediate property* of an entity is a single-valued piece of information that describes an entity in some way. For example, "name" and "salary" are both immediate properties of the entity type "Teacher". Immediate properties are represented in *P-relations* which have as their primary key an E-attribute whose main function is to link each immediate property with

the entity it is describing in the E-relation. Insertions into P-relations and deletions from E-relations are governed by the following *Property Integrity* rule:

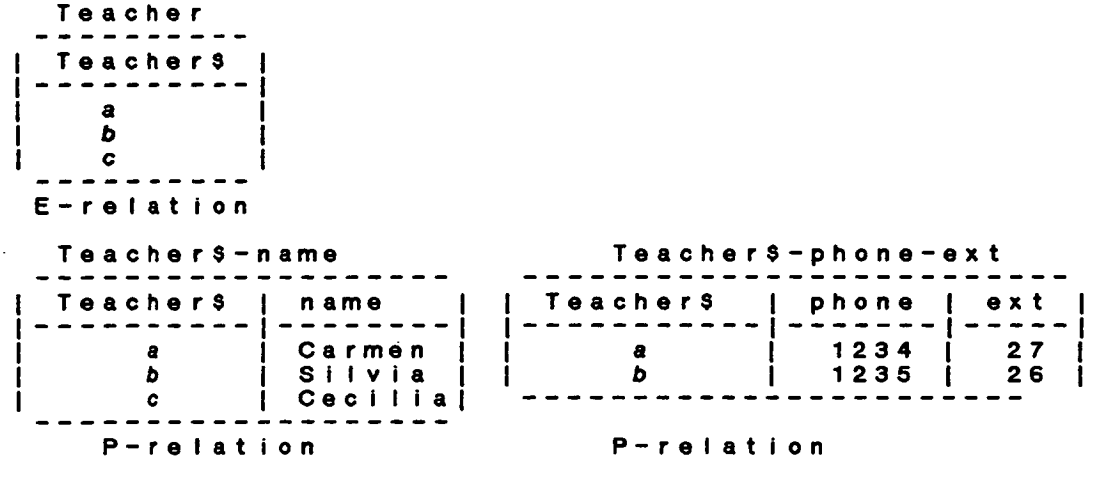
> *"if a tuple t appears in a P-relation P, then the (surrogate) primary key of t must appear in the E-relation corresponding to P"*

As a convention, an E-relation is named after its entity type name (e.g. "Teacher", "Student") and its E-attribute after the E-relation name concatenated with the character "$" (e.g. "Teacher$", "Student$"). Examples of E and P relations are shown in figure 4-3.

The user may or may not be aware that a "Teacher" is identified by a surrogate (*a*, *b* or *c*). This may depend on the application or the sophistication of the user. RM/T is intended to be primarily of interest to the database designer i.e. as a basis for designing the conceptual schema of the system. As shown in figure 4-3, a P-relation may be of any degree > 1. The number of attributes in a P-relation is left to the discretion of the database designer. At one extreme, each P-relation is a binary relation containing the values of a single immediate property. At the other extreme, all attributes are grouped in one P-relation. Consider for example the attribute "telephone". It is the designers choice to consider the phone number and the extension as atomic units and define two separate P-relations, or have them all together in one as shown in figure 4-3.

We make an additional simplification: an E-relation and all its P-attributes are stored in one single relation. This is called a *compressed* E-relation and is facilitated by the use of default values in those P-attribute values that have not been defined e.g. "phone" and "ext" of "Cecilia".

Figure 4-3: E and P relations in RM/T



The P-relations contain the actual database data, the extension of the database. Other relations which will be defined next constitute the meta-data and are represented in various *Graph* relations. As a convention, we name each P-relation after its immediate properties, separated by the "-" character e.g. Teacher$-phone-ext.

A binary relation called the *Property Graph* relation PG indicates which P-relations contain properties of which E-relation. It contains a tuple for every P-relation in the database whose attribute values are the name of a P-relation and the name of its associated E-relation. Figure 4-4 shows the PG relation, and some of its tuples, for the database in figure 4-1.

Figure 4-4: Property Graph Relation PG

PG

P-relation	E-relation
Teacher$-t#	Teacher
Teacher$-name	Teacher
Teacher$-phone-ext	Teacher
Student$-name	Student
.....	.....

4.4.2 Multivalued and Indirect Properties

Multivalued properties are equivalent to "repeating fields" in records, e.g. the "job history" of a teacher. Multivalued properties may themselves have multivalued properties such as the "salary history" of each "job history" entity.

Indirect properties (or transitive functional dependencies) arise when an immediate property itself has another immediate property e.g. a "blood" entity which has one of many "blood-types" which in turn is either "positive" or "negative".

Multivalued properties and an immediate property having itself immediate properties are classified as *Characteristic* entities in that they fill a subordinate role in describing or qualifying entities of another type.

The *Characteristic Graph* relation CG indicates which entity types are characterised by which other types. It is a binary relation with one attribute having the subordinate role ("Sub") and the other attribute the superior role ("Sup"). Each characteristic entity type has its E-relation and a (possibly empty) set of P-relations. An example is shown in figure 4-5.

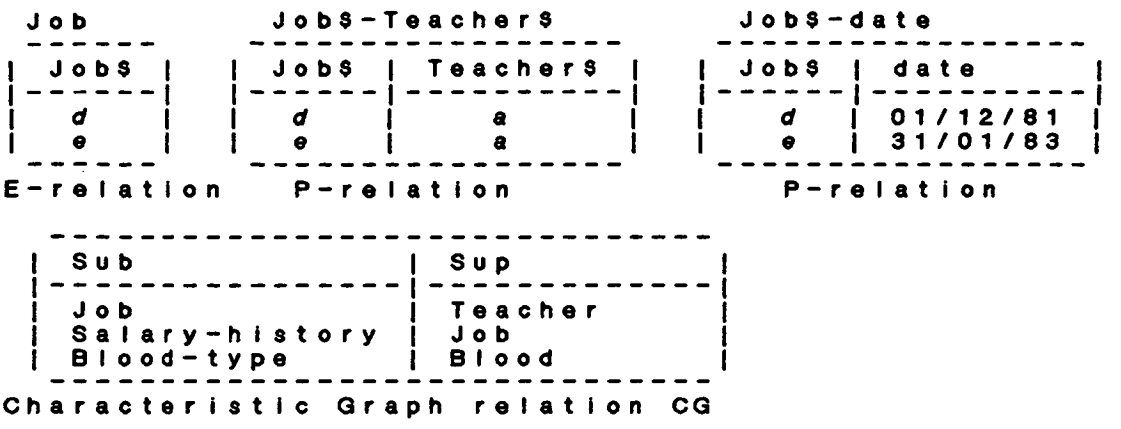
Insertion and deletion of characteristic entities are governed by the following *Characteristic Integrity* rules:

> *"A characteristic entity cannot exist in the database unless the entity it most immediately describes is also in the database."*

In other words, this integrity rule states that a characteristic entity is *existence-dependent* on the entity it is describing.

In the original relational model, there is no way other than looking at the actual attribute values or through explicit integrity constraints, to know whether a given attribute that is not defined as a primary key is of a characteristic type.

Figure 4-5: Characteristic Graph Relation



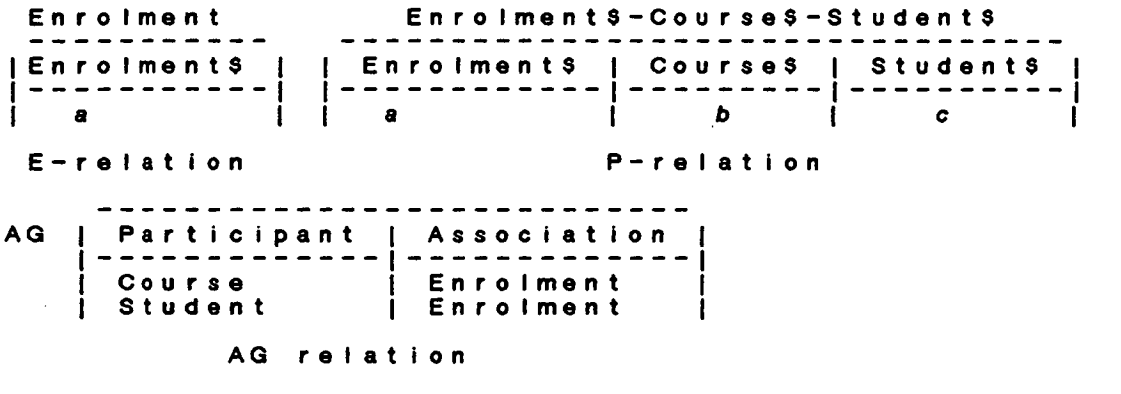
4.4.3 Associative Entities

Entities are classified as *Associative* if they fill a superior role in interrelating entities of possibly different types, called the *immediate participants* of the association. Unlike the characteristic case, entities that are immediate participants in an association are not existence-dependent on one another (e.g. "Course" and "Student" entities associated through "Enrolment" entities).

Associations are accorded the status of entities and therefore represented by an E-relation and a set of P-relations.

The *Associative Graph* relation **AG** is a binary relation that indicates which entity types participate in which associations. It contains a tuple of the form (I,A) for every E-relation I that immediately participates in the definition of the associative type A. An example is shown in figure 4-6.

Figure 4-6: Associative Graph Relation



Note that the binary relation "Attend" in figure 4-1 had to be created to establish associations between courses and students. Attributes such as "s#" and "c#" were created for the sole purpose of establishing a link between these relations. (The same arguments apply to the relation "Organise", "Workfor" and "Belong"). In RM/T the user is neither forced to create such artificial attributes nor any artificial "link" relation, such as "Attend". It is the AG relation which will explicitly contain information for the different associations existing in the database.

In the example in figure 4-6 the P-relation has E-attributes only and its tuples consist of surrogate values. This is not necessarily always so. The association may be established between more than two relations and it can have immediate attributes itself (e.g. date of enrolment).

Associative entities support one type of *Cartesian Aggregation* abstraction [Smith 77] (the other types are supported by the P together with the PG relations and the characteristic relations together with the CG relation). A cartesian aggregation is an abstraction in which the relationship between entities is regarded as a higher level entity.

Insertion, deletion and updates of associative entities are governed by the following *Associative Integrity* rule (without considering null values):

> *"let A be an associative entity type, and let E be the set of E-attributes whose function is to identify the participants in A. Then a given instance of A can exist in the database only if, for that instance, each E-attribute in E identifies an existing entity of the appropriate type."*

The existence of the Characteristic and Associative meta-relations should make the task of understanding how a set of relations is modeling an application easier than in the original relational model.

4.4.4 Kernel Entities

A *kernel* entity is an entity that is neither associative nor characteristic e.g. "Student", "Course", "Department" and "Teacher".

4.4.5 Generalization

Generalization is an abstraction in which a set of similar objects is regarded as a generic object. It embodies the concepts of subtype (or specialization) and supertype (or generalization).

An entity type e1 is said to be a *subtype* of entity type e2 if all entities of type e1 are also of type e2. For example, "Student" and "Teacher" may both be subtypes of "Person". All the properties that apply to an entity type also apply to their subtypes (the converse is not true).

An entity type, its immediate subtypes and their immediate subtypes, and so on, constitute a *generalization hierarchy*. The *Generalization Graph* relation GG is used to represent the generalization hierarchies. A hierarchy may be divided into many different sub-hierarchies and is therefore organised in categories. The GG relation is therefore a ternary relation containing a tuple for every subtype, its immediate supertype and the corresponding category. An example is shown in figure 4-7.

Figure 4-7: Generalization Graph Relation

GG

Subtype	Supertype	Category
Student	Person	Role
Teacher	Person	Role
Teacher	Employee	Job
Secretary	Employee	Job

The insertion and deletion of entities in a generalization hierarchy is governed by the following *Subtype Integrity* rule:

> *"Whenever a surrogate s belongs to the E-relation for an entity type e, s must also belong to the E-relation for each entity type of which e is a subtype.*

Any entity type (characteristic, kernel or associative) may have one or more subtypes, which in turn may also have subtypes. A subtype of a characteristic entity type is also characteristic, a subtype of a kernel entity type is also kernel and a subtype of an associative entity type is also associative.

Generalization hierarchies offer the possibility of having different levels of access to a database. Consider for example a car dealer who organises his cars in a generalization hierarchy that has commonplace facts about cars at the top of the hierarchy e.g. those of interest to salepersons and customers, and more specialised facts further downstream e.g. those of interest to car repairers. Another advantage of generalizations is the enhancement of the stability of an application's schema where the changes at one level of the hierarchy do not affect levels above it. Generalization and Aggregation are considered important abstraction tools in data modeling [Smith 77].

4.4.6 Operations in RM/T

Some of the operations defined in RM/T that are not included in the original relational model are described next.

OUTER JOIN. The join operation of the relation model has the property of not including tuples in the result relation if a tuple in one of the argument relations has no matching value (in the attributes used in the join operation) in the other argument relation. There is therefore a potential loss of information for unmatched tuples. The outer join is defined as an *information preserving* operation in that it includes non matching tuples in the result relation by inserting default values into the corresponding attributes of the other relation.

Consider the P-relations shown in figure 4-3. To produce a relation that contains all the immediate properties of the entity type "Teacher", we can outer join its P-relations to produce the relation shown in figure 4-8. In that relation, we assume that the default value for the attributes "phone" and "ext" is "0".

Figure 4-8: Result of Outer Join

Teacher$	name	phone	ext
a	Carmen	1234	27
b	Silvia	1235	26
c	Cecilia	0	0

For the purposes of our research, we defined an *UNIDIRECTIONAL* outer join (UOJ), similar to that defined in [Smith 80]. If R and S are two relations, then R UOJ S includes only non-matching tuples of the left-hand side argument relation (R). In other words, the result relation will contain at least one occurrence of each tuple of the left-hand side relation in the UOJ operation. Its use is explained in a later section and its definition is the following:

```
R UOJ S = { (r,s) |
((r is in R) and (s is in S)  and (r.ja = s.ja)) or
((r is in R) and (s is in SD) and
  (there does not exist s1 in S | r.ja = s1.ja)) }
```

where *ja* is the join attribute and SD is a relation of cardinality 1 with identical attributes to those in S and whose single tuple has default values only.

JOIN operations are defined, in fact, as *theta* joins. The join attributes are not only tested for equality but for any other (theta) comparison operator defined for the domain of the join attributes (e.g. >, <, etc.)

The NOTE operation produces the name (character string) of a relation e.g. if R is a relation then NOTE(R) is the name of R.

DENOTE and NOTE are inverses of one another: if r is the name of a relation R, then DENOTE(r) is the relation denoted by r.

COMPRESS is an operator that has a commutative operation associated with it and operates over sets of relations. When applied to a set of relations, it returns a single relation, obtained by repeatedly replacing pairs of relations in the set by a single relation (according to the operation associated with the compress operator), until only one relation is left. Consider for example a relation containing the name of all P-relations of a given entity type e. The compress operator, associated with the outer join operator (defined over the common E-attribute) would return a relation containing all immediate properties of the entity type e.

The compress and outer join operators may be used to provide views of a relation in terms of their properties and have been combined in one operator called PROPERTY. Hence the PROPERTY operator takes an entity type and returns a relation having its attributes taken from

all the P-relations associated with the entity type, regardless of how many properties there are, how these properties are grouped into P-relations or how the P-relations are named.

APPLY is an operator that takes as arguments an operator f and a set X and produces as result the set of all objects f(X) obtained by applying f to every member of X. Consider for example X as a unary relation containing each tuple the name of an entity type (E-relation). The operation APPLY(PROPERTY,X) returns a set of relations, each one corresponding to one entity type with all its P-relations.

Many other new operations have been defined in [Codd 79].

4.4.7 RM/T Catalog

The RM/T Catalog contains the **relations**, **attributes** and **domains** relations. These three relations contain all the necessary information about all the relations, attributes and domains defined in the database. Operations over these relations have special side-effects which are controlled by the underlying DBMS (e.g. adding a tuple to the **relations** relation causes the system to create a new relation in the database). Some of the tuples contained in these relations, according to previous examples, are shown in figures 4-9 and 4-10. To simplify the examples, we have not considered attribute domains, instead the relation "attributes" is shown with the "att-type" attribute.

Figure 4-9: Catalog Relation "relations"

rel-name	rel-type
relations	Catalog
attributes	Catalog
AG	Graph
CG	Graph
GG	Graph
PG	Graph
Teacher	Kernel
Teacher$-name	Property
Teacher$-phone-ext	Property
Job	Characteristic
Job$-Teacher$	Property
Job$-date	Property
Enrolment	Associative
Enrolment$-Course$-Student$	Property

4.5 Inter Model Mapping

Any system dealing with heterogeneous data models has the problem of defining a common conceptual framework in which equivalence properties can be meaningfully established. Borkin offers a formalism for analysing data model equivalence [Borkin 78]. Multibase [Smith 80] aims to integrate existing heterogeneous distributed databases, offering one unifying view (a

Figure 4–10: Catalog Relation "attributes"

rel-name	att-name	att-type
relations	rel-name	String
relations	rel-type	String
attributes	rel-name	String
attributes	att-name	String
attributes	att-type	String
AG	Participant	String
AG	Association	String
CG	Sub	String
CG	Sup	String
PG	P-relation	String
PG	E-relation	String
GG	Subtype	String
GG	Supertype	String
GG	Category	String
Teacher	Teacher$	Surrog
Teacher$-name	name	String
Teacher$-phone-ext	phone	Integer
Teacher$-phone-ext	ext	Integer
Job	Job$	Surrog
Job$-Teacher$	Job$	Surrog
Job$-Teacher$	Teacher$	Surrog
Job$-date	Job$	Surrog
Job$-date	date	date
Enrolment	Enrolment$	Surrog
Enrolment$-Course$-Student$	Enrolment$	Surrog
Enrolment$-Course$-Student$	Course$	Surrog
Enrolment$-Course$-Student$	Student$	Surrog

global schema) of the entire database to give the illusion of a non–distributed homogeneous system. Various levels of schemas facilitate the decomposition of a query over the global schema into queries over local schemas at possibly different sites. Each site may have its portion of the database expressed in some data model and implemented in some database system different from another site.

The Multibase approach is essentially the inverse of ours. By advocating the *external* multimodel approach, we are offering an architecture for systems that *evolve* into heterogeneous systems. The class of problems that Multibase aims to alleviate are therefore different. There are, however, some points in common considering that both approaches will have to map requests from one model into another.

We perceive our class of problems as being relatively simpler than those of Multibase. First, the availability of a common ICS and a canonical IQL and IDML makes the management of physically distributed databases simpler. Consider a distributed database where each site implements the common IQL, IDML and ICS under the same principles. Provided a global internal conceptual schema exists to which each site can refer, requests expressed in the common IQL or IDML are easily portable. Second, we consider an ICS capable of expressing a wide variety of integrity constraints associated with a single "Constraint Handler" (common to all external systems) that enforce the integrity rules. This guarantees and facilitates the preservation of consistent database states. (This localisation of tasks is another advantage of internal database standards which is discussed in more detail in chapter 6).

What we offer may be considered as an internal database system and a set of syntactically different external systems.

THE PROBLEMS OF INTER MODEL MAPPING.

A multi-model such as this poses many questions regarding the mapping of objects between the internal conceptual schema (ICS) and the external schemas. Although the semantic expressiveness of data models can extend into an indefinite continuum, starting from the von Neuman store, we will distinguish between two levels of semantic capabilities provided by the coexisting models to resolve those questions. We will refer to those models having the structural and operational capabilities of the original relational model as *basic* models (e.g. SQL, QBE) and those at the level of the RM/T as *advanced*. Our architecture permits users to start at the basic level and advance gradually upwards. The implementation of the language RAQUEL (see chapter 7) represents an intermediate stage.

The mapping problems may be classified as structural, operational, name handling, constraint handling and I/O handling. Most of the questions presented next are answered throughout the rest of the chapter. Some unresolved problems are highlighted in the conclusions.

1. STRUCTURES:

a) How do *basic* and *advanced* systems coexist i.e. how are advanced structures seen from a basic system. The inter-level mapping poses the problem of representing an advanced structure (e.g. type hierarchies) in a basic system. We assume that any structure of a basic system is directly reflected on an advanced system i.e. a basic system is a subset of an advanced system.

b) How to transform an existing basic system into an advanced system i.e. the existing database (ICS) was created through a basic query language and it is desired to upgrade to a system offering advanced capabilities.

c) How to produce an external conceptual schema for a non-relational advanced system. This case will be analysed in detail for the functional model and the language EFDM [Kulkarni 83] presented in the next section.

2. OPERATIONS:

a) How to map advanced non-relational queries into the common internal query language. Again, this case will be analysed for EFDM.

b) How to map update operations into the common data manipulation language, enforcing integrity constraints.

3. NAME HANDLING:

How to provide a suitable naming system for each external model e.g. functional systems may regard the database as consisting of "objects" rather than relations. The problems of view handling and view mapping is also relevant and it has been addressed only for EFDM.

4. CONSTRAINTS:

a) What is the class of constraints supported implicitly and explicitly by the ICS and the underlying DBS.

b) How are constraints to be enforced e.g. some implicit ICS constraints due to the classification of entities into types may not be apparent in a basic system

c) How to translate non-relational integrity constraints into ICS constraints. This is more a matter of syntactic transformation into the ICS constraint specification formalism.

5. I/O

a) How to deal uniformly with the layout of results, re-using if possible common internal output procedures.

b) How to deal uniformly with database loading from external files.

We restrict the answer to these problems to the cases involving the functional data model and the language EFDM, the relational model and the "semi" advanced query language RAQUEL and the basic query language TABLES, presented in chapter 7.

We will decompose each external conceptual schema into two components. One is the *user-conceptual schema* (UCS) and the other is the *mapping schema* (MS). The former is the conceptual schema as presented to the users, it contains the objects that are visible to them (on top of it several external schemas may coexist) and the latter contains those same objects but translated into RM/T relational operators and in terms of the corresponding objects in the ICS. [Stocker 81] describes a similar approach.

4.6 The Functional Query Language EFDM

The Functional Data Model and one of its representatives, the data definition and manipulation language Daplex proposed in [Shipman 81] was chosen for experimenting with the multi-model approach. This decision was taken due to first, the interest in testing the suitability of RM/T to support a non-relational external world that would enrich the choice of models and languages offered at the user level, second because an extended version of it has been implemented at the department in a system called EFDM (Extended Functional Data Model) by Kulkarni [Kulkarni 83] which has given the opportunity to discuss conceptual as well as implementational aspects and third, because Daplex is the subject of a heterogeneous system implementation at CCA [Chan 81] providing a basis for comparison.

It is claimed in [Shipman 81] that Daplex, because of its functional notation, provides a "conceptually natural" interface to database systems in that

> *"it allows the user to more directly model the way he thinks about the problems he is trying to solve".*

In other words, Daplex is claimed to be a conceptually concise language in that

> *"it reduces the need for introducing artificial elements when formally specifying a query".*

Preliminary observations by Kulkarni at demonstrations in the department to students, visitors and others suggest this is a valid claim.

The following presentation considers a simplified and modified version of EFDM and the mapping procedures from the external EFDM schema to the ICS. Some operations obviously have an effect on the internal schema and the database itself but this will depend on the particular implementation of the system and is therefore not discussed here (e.g. when adding a new attribute to a relation of cardinality > 0, the underlying DBS may extend the tuples of the relation by adding default values and create an index to the new attribute. These operations will be recorded in the common internal schema).

Recall that every EFDM schema has two components, the *user-conceptual* schema (UCS) and the *mapping* schema (MS). An EFDM interface is provided with the rules to translate objects in the UCS to objects in the MS expressed in terms of objects in the ICS. A MS object is standardised and therefore understood and subject to interpretation by the underlying database software. The following sections present the rules that govern the mapping of objects between the UCS to the MS and ICS. Although the effects on the ICS and MS are shown at the level of "compressed" relations i.e. E-relations which have been "outer joined" with its P-relations to form one semantic unit, the extension of these effects to individual P-relations is easily achieved. The assumptions of compressed relations facilitates the translation of EFDM statements (e.g. no Property Graph relation is needed) and it is the underlying system which is left in charge of decompressing them.

The syntax of relational expressions is hopefully self-explanatory, and we choose the one used in RAQUEL (see chapter 7 for details). The examples are drawn from the database shown in figure 4-11 which is a modified version of figure 4-1. Entity types are enclosed in boxes and arrows depict functions mapping their argument types into their result types. Single headed arrows denote single-valued functions, double-headed denote multi-valued functions.

EFDM models applications in terms of entities (objects) and functions (properties of the objects or associations among them).

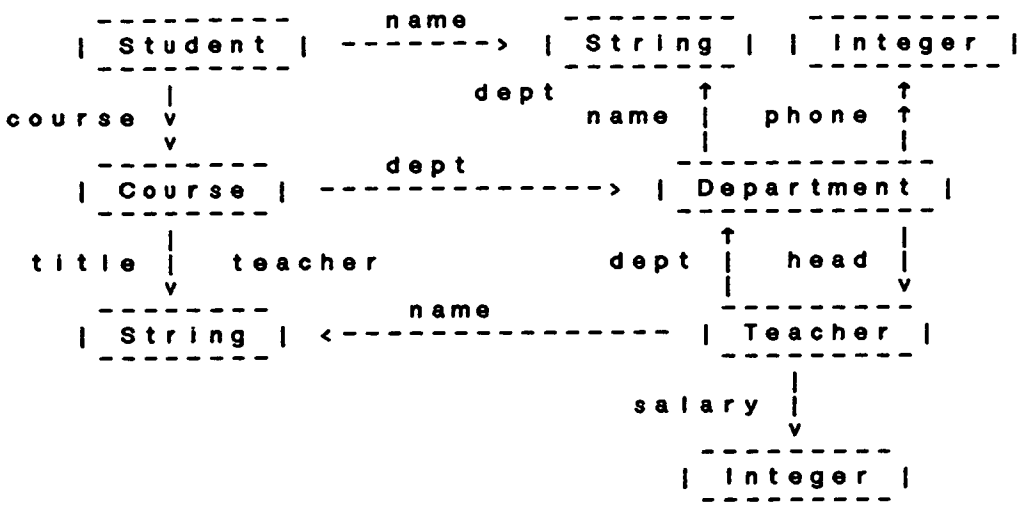
4.6.1 EFDM Data Definition: DECLARE

Two forms of data definition are provided in EFDM. One is the DECLARE statement, used to establish both entity types and primitive functions in the system which are directly updatable. The other form is the DEFINE statement, used to establish derived functions by composing existing functions. We look at each in turn.

The syntax of EFDM is defined according to the following rules: repetition is denoted by curly brackets, optionality is expressed by square brackets and parenthesis is used for grouping. Terminal symbols are written in bold face and non-terminals in bold italic face.

The syntax of the DECLARE statement (expressed in UCS terms) is:

Figure 4-11: EFDM University Database



DECLARE ***functionid*** **(** ***arglist*** **) [-> | ->>]** ***typeid***

functionid is a function identifier, ***arglist*** is an argument list and ***typeid*** is a type identifier. Single headed arrows specify single-valued functions and double-headed arrows specify multi-valued functions. Unless otherwise stated, the following presentation considers single argument functions only.

The effect of the DECLARE statement on the MS and ICS depends on the argument list and the type identifier. The following cases may arise:

1) Declare a new (kernel) entity type.

This case occurs if the ***arglist*** is empty and the ***typeid*** is **Entity**:

e.g. DECLARE Student() ->> Entity

Entity is the system provided type for all *kernel* entities. It corresponds to the kernel relations in RM/T. If the argument list is empty then it implicitly declares a type identifier. Thus, the above example is used to declare both the function "Student" and the entity type "Student". The effects of the above declaration on the ICS are:

a) Create a new kernel type relation i.e. an E-relation with its E-attribute.

b) Insert a tuple into the RM/T Catalog relation "relations" asserting the existence of the new E-relation (e.g. Student).

c) Insert a tuple into the RM/T Catalog relation "attributes" asserting the existence of the new E-attribute (e.g. Students).

This declaration has no effect on the MS since a kernel entity type has an immediate correspondence with a kernel relation.

2) Declare an immediate property of an entity type.

The informal definition of immediate properties given in section 4.4.1 is formalised as:

A function over an entity type e is said to be an *immediate property* of the entity type e if it is single valued and its result type is one of the system defined scalar types such as "String", "Integer" or "Boolean".

e.g. DECLARE name(Student) -> String

The previous statement defines the mapping of entities of type "Student" to entities of type "String". The effects on the ICS, after ensuring that the argument exists as an E-relation and the property has not been already declared for that entity type, are:

a) Append the new property as an attribute to the corresponding E-relation. The ***functionid***'s name dictates the name of the new attribute ("name").

b) Insert a tuple into the "attributes" relation asserting the existence of the new attribute and indicating its type.

If we had considered P-relations instead of compressed relations, then the following procedure would have been performed instead:

a.1) Create a binary P-relation. One of its attributes is the E-relation's E-attribute ("Student$"). The other attribute is named after the ***functionid***'s name ("name"). The new P-relation is, by our earlier convention, named "Student$-name".

b.1) Insert a tuple into the Property Graph relation (PG) linking the new P-relation ("Student$-name") with its E-relation ("Student").

c.1) Insert two tuples into the Catalog relation "attributes" asserting the existence of the two new attributes ("Student$" and "name").

There is no effect on the MS since an immediate property corresponds directly to a P-relation (or to an attribute of the compressed E-relation). Note that if the internal DBS has no P-relations but compressed E-relations, then the addition of an immediate property causes the addition of an attribute to it, plus the extension of the existing tuples with default values. Since EFDM does not define default values in function declarations, these are taken from the underlying system defined default values (e.g. "" for strings, "0" for integers and "false" for booleans", although any other could be defined). If the underlying system has P-relations then the MS could maintain the compressed view through the PROPERTY operation, as defined in section 4.4.6.

3) Declare a multivalued property of an entity type.

A multivalued property of an entity type corresponds to a characteristic entity in RM/T:

e.g. DECLARE phone(Department) ->> Integer

The double arrow indicates that it is a multi-valued function which, when applied to entities of type "Department", returns a (possibly empty) set of entities of type "Integer". The effects of the above declaration on the ICS are:

a) Create a (compressed) ternary Characteristic E-relation. It is named after the entity type it is describing, specified in the ***arglist*** ("Department"), and the ***functionid***'s name ("phone"): "Department-phone". Its attributes are: its E-attribute ("Department-phone$"), the E-attribute of the entity it is describing ("Department$") and the new characteristic property name ("phone"):

b) Insert a tuple into the Characteristic graphs relation (CG) indicating the "Sub" role of the E-relation and the "Sup" role of the ("Department-phone" and "Department")

c) Insert the necessary tuples into the Catalog relations "relations" and "attributes" asserting the existence of the new relation and attributes.

Given that a characteristic property is represented in a different relation than the entity type it is characterising, it is convenient that the MS contains the operations that will connect them. Using the present "Department" and "phone" example, this means that we need to define a relational expression that, given a department entity will yield its phone numbers. This is achieved by defining an unidirectional outer join (UOJ) between the E-relation and its characteristic relation, through the common E-attribute e.g.:

Department UOJ THROUGH Department$ = Department$ TO Department-phone

The need for an outerjoin instead of a simple join arises from the fact that a join existencially qualifies the tuples of the relations it is joining so that a request involving joins on departments and their phones may result in a relation not containing those departments which do not have phones.

The general expression connecting entity types and their characteristic properties is:

arglist UOJ THROUGH e-att = e-att TO ***arglist*** ++ "-" "" ***functionid***

Where ***arglist*** is the argument of the function (recall that we are considering the single argument case only) and e-att is the common E-attribute's name.

4) Declare a new subtype entity type:

Instead of declaring "Student" as of type "Entity", it is possible to declare it as a subtype of another already existing type. Assume that the following declaration exists in the UCS:

```
DECLARE Person() ->> Entity
```

"Student" could then be declared as a *subtype* of the *supertype* "Person":

```
DECLARE Student() ->> Person
```

Subtypes inherit the functions declared over their supertypes, thus if the type "Person" has the function "name" over it, then the application of "name" over entities of type "Student" is a valid operation. In fact, all entity types are subtypes of the system provided "super" type "Entity". This mechanism organizes types into hierarchies and implements the generalization abstraction. The effects on the ICS, after ensuring that the supertype exists, are:

a) Create a new E-relation, named after the ***functionid***'s name ("Student"). Note that the domain of this attribute is a subset of the domain of the corresponding E-attribute of the supertype.

b) Insert a tuple into the generalization graph relation (GG) with the ***functionid***'s name as the "Sub" attribute value and the ***typeid*** as the "Sup" attribute value ("Student", "Person").

c) Insert the necessary tuples into the Catalog relations "relations" and "attributes" asserting the existence of the new relation and attribute.

The effect on the MS is the definition of a relational expression linking the subtype with the supertype entities. In this case it is the join operation (and not outerjoin) which is used since we are interested in the subtype entities and not in the possibly larger collection of all supertype entities. The relational expression is:

functionid JOINED THROUGH e-att1 = e-att2 TO ***typeid***

e-att1 is the E-attribute of the subtype ("Student$") and e-att2 is the E-attribute of the supertype ("Person$"). The example would be coded as:

```
Student JOINED THROUGH Student$ = Person$ TO Person
```

5) Declare an associative entity type:

If the ***arglist*** is not empty and the ***typeid*** is not of a scalar type, then the declaration establishes an association between the entity types. Associations may be established between entities of the same type. The following example connects students and the department to which they belong. The next example associates entities of the same type.

```
DECLARE dept(Student)      -> Department
DECLARE friends(Student) ->> Student
```

The "dept" function, when applied to a "Student" entity returns a "Department" entity (and not a "Department" number or any other property of "Department"). The effect on the ICS, after ensuring that the involved entity type in the ***arglist*** exists is:

a) Create a (compressed) ternary Associative E-relation. Its name is formed by concatenating the ***functionid*** and the ***arglist*** ("dept-Student"). Its attributes are its own E-attribute ("dept-Student$") and the E-attributes of the types involved in the declaration ("Department" and "Student").

b) Insert the necessary tuples into the Association Graph relation (AG) with the "Sub" attribute taking its value from the type identifiers and the "Sup" attribute from the new associative relation name e.g. ("Student","dept-Student") and ("Department","dept-Student").

c) Insert the necessary tuples of the new relations and attributes into the Catalog relations "relations" and "attributes".

The effect on the MS is to define a relational expression linking the participant entity types:

```
arglist UOJ THROUGH e-att1 = e-att2 TO assoc-name
        UOJ THROUGH e-att1 = e-att2 TO typeid
```

where e-att1 is the E-attribute's name of the type in the ***arglist***, e-att2 is the E-attribute's name of the type in the ***typeid*** and assoc-name is the associative relation's name. The example would become:

```
Student UOJ THROUGH Student$=Student$ TO
   dept-Student UOJ THROUGH
      Department$=Department$ TO Department
```

The fact that the relationship between students and departments is 1:n can be enforced by defining integrity constraints in the Associative relation: the "Student$" attribute value may be enforced to be "unique" or "key" (this will also enforce that a student belongs to at most one department).

6) Declare a multiargument function.

This and the previous case are similar. The ***arglist*** and the ***typeid*** entity types establish an association among them e.g.

```
DECLARE grade(Course,Student) -> String
```

The difference lies in the fact that the ***typeid*** "String" is one of the system defined types and therefore an association is established only between the ***arglist*** types. The effect on the ICS is as follows:

a) If an Associative relation already exists between the entity types, then insert into it the attribute corresponding to the ***functionid***'s name ("grade"), otherwise create a new association relation as in the previous case (including all the E-attributes of the entity types in the ***arglist***).

b) Insert the corresponding tuples into the AG relation, as before.

c) Insert the necessary tuples into the Catalog relations "relations" and "attributes", as before.

The situation is similar for declarations involving more than two arguments and the ***typeid*** different from any system defined scalar type. It involves the definition of outerjoin expression in the MS to link the types together. A difference exists if the ***typeid*** is of type "Entity", in which case the link is established between the ***arglist*** types only e.g.:

```
DECLARE enrolment(Course,Student) -> Entity
```

To illustrate the practical application of these rules consider the well known students, teachers and courses example which was shown diagramatically in figure 4-11 and which would be represented in an EFDM user conceptual schema as shown in figure 4-12. The corresponding RM/T MS and ICS schemas and the Catalog relations "relations" and "attributes" (meta-data tuples not shown) are shown in figures 4-13, 4-14, 4-15 and 4-16. The kernel type relations in figure 4-13 contain the actual database data, which is not shown.

Figure 4–12: EFDM User Conceptual Schema

```
DECLARE Student() ->> Entity
DECLARE name(Student) -> String
DECLARE dept(Student) -> Department
DECLARE course (Student) ->> Course

DECLARE Teacher() ->> Entity
DECLARE name(Teacher) -> String
DECLARE salary(Teacher) -> Integer
DECLARE dept(Teacher) -> Department

DECLARE Department() ->> Entity
DECLARE name(Department) -> String
DECLARE phone(Department) ->> Integer
DECLARE head(Department) -> String

DECLARE Course() ->> Entity
DECLARE title(Course) -> String
DECLARE teacher(Course) -> Teacher
```

Figure 4–13: RM/T Internal Conceptual Schema

Compressed Kernel relations Student and Department

Student$	name

Department$	name	phone

Compressed Kernel relations Course and Teacher

Course$	title

Teacher$	name	salary

Characteristic Graph relation CG

Sub	Sup
phone-Department	Department

Associative Graph relation AG

Participant	Association
Student	dept-Student
Department	dept-Student
Student	course-Student
Course	course-Student
Department	dept-Course
Course	dept-Course
Teacher	teacher-Course
Course	teacher-Course
Teacher	dept-Teacher
Department	dept-Teacher
Department	head-Department
Teacher	head-Department

Figure 4–14: RM/T Mapping Schema

```
phone(Department):Department UOJ THROUGH
 Department$=Department$ TO Department-phone

dept(Student):Student UOJ THROUGH
 Student$=Student$ TO dept-Student UOJ THROUGH
  Department$=Department$ TO Department

course(Student):Student UOJ THROUGH
 Student$=Student$ TO course-Student UOJ THROUGH
  Course$=Course$ TO Course

dept(Course): Course UOJ THROUGH Course$=Course$
 TO dept-Course UOJ THROUGH Department$=Department$
    TO Department

teacher(Course):Course UOJ THROUGH Course$=Course$
 TO teacher-Course UOJ THROUGH Teacher$=Teacher$
    TO Teacher

dept(Teacher):Teacher UOJ THROUGH Teacher$=Teacher$
 TO dept-Teacher UOJ THROUGH
   Department$=Department$ TO Department

head(Department):Department UOJ THROUGH
 Department$=Department$ TO head-Department
   UOJ THROUGH Teacher$=Teacher$ TO Teacher
```

Figure 4–15: RM/T Catalog relation "relations"

rel-name	rel-type
Student	Kernel
Course	Kernel
Teacher	Kernel
Department	Kernel
dept-Student	Associative
course(Student)	Associative
dept(Course)	Associative
dept(Instructor)	Associative
head-Department	Associative
phone-Department	Characteristic

4.6.2 EFDM Data Definition: DEFINE

DEFINE is a mechanism for deriving associations among existing entity types without creating new objects in the ICS. It is a useful tool for creating abstractions which can be further used in building higher-level abstractions. This ability is much appreciated when a complex request can be constructed out of simpler ones or as shorthand notation for lengthy and repetitive expressions. This facility may greatly enhance the appeal of a user interface, and be used to define views significantly different from the stored data.

Figure 4–16: RM/T Catalog relation "attributes"

rel-name	att-name	att-type
Student	Student$	Surrog
Student	name	String
Course	Course$	Surrog
Course	title	String
Teacher	Teacher$	Surrog
Teacher	name	String
Teacher	salary	Integer
Department	Department$	Surrog
Department	name	String
dept-Student	dept-Student$	Surrog
dept-Student	Student$	Surrog
dept-Student	Department$	Surrog
course-Student	course-Student$	Surrog
course-Student	Student$	Surrog
course-Student	Course$	Surrog
dept-Course	dept-Course$	Surrog
dept-Course	Course$	Surrog
dept-Course	Department$	Surrog
teacher-Course	teacher-Course$	Surrog
teacher-Course	Course$	Surrog
teacher-Course	Teacher$	Surrog
dept-Teacher	dept-Teacher$	Surrog
dept-Teacher	Department$	Surrog
dept-Teacher	Teacher$	Surrog
head-Department	head-Department$	Surrog
head-Department	Department$	Surrog
head-Department	Teacher$	Surrog
phone-Department	phone-Department$	Surrog
phone-Department	Department$	Surrog
phone-Department	phone	String

The DEFINE statement may only affect the UCS which stores the definition of the statement and the MS which stores the equivalent relational expression which is used directly when translating the query. Alternatively, only the UCS may store the definition which can be dealt with by substitution into any ICS query using it. The former approach may save translation time in complex queries involving many previous definitions. The syntax of the DEFINE form of data definition is:

DEFINE *functionid* (*arglist*) [-> | ->>] *set*

Where *set* is any valid expression yielding a set i.e. an entity type e.g. define an EFDM expression that applied to "Course" entities yields a set of "ART" courses:

```
DEFINE Art(Course) ->> Course
   SUCH THAT dept(Course) = "ARTS"
```

The mapping of DEFINE statements is essentially similar to the mapping of EFDM queries, presented in the next section.

4.6.3 EFDM Queries

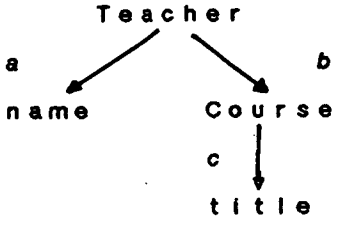
EFDM queries consist of "FOR" loop statements, nested functions, predicates and terminate with a "PRINT" statement. A query that retrieves the titles of the courses taught by the teacher named "Carmen" can be expressed as:

```
FOR EACH Teacher SUCH THAT
  name(Teacher) = "Carmen"
    FOR EACH Course(Teacher)
      PRINT title(Course)
```

The FOR EACH statement iterates over a set of entities of type "Teacher" which are filtered by the predicate following the SUCH THAT clause. The PRINT statement indicates which property values are to be displayed. Note that the functions to be printed must yield printable (scalar) values (e.g. strings, integers, etc.).

EFDM queries can be represented as a directed graph and a *PRINT-list* indicating the functions that participate in the PRINT statement. In what follows we present the structure of query graphs for single-argument functions and a method of deriving a relational expression semantically equivalent to the query represented in the graph. The presentation is based on the previous example which can be represented in the graph shown in figure 4-17.

Figure 4-17: Query Graph 1



A graph consists of *nodes* and *arcs* connecting the nodes. The *root* node of the graph corresponds to the entity type present in the first FOR loop (e.g. "Teacher"). Arcs connect functions with their arguments. The "function" node is called the *head* node and the "argument" node the *tail* node. If a node is both head and tail (e.g. "Course") then it is a *non-terminal* node, otherwise it is a *terminal* node (e.g. "name" and "title").

An arc is labelled with the "partial" relational expression representing the link between its two nodes. It is partial because the expression is later completed when adding the node's entity type to the expression e.g. *a* is the partial select operation SELECTED ON name = "Carmen", which is later becomes: Teacher SELECTED ON name = "Carmen".

c is a partial project operation: PROJECTED TO title.

b is the partial relational expression defined in the MS between "Teacher" and "Course": UOJ THROUGH Teacher$=Teacher$ TO course-Teacher UOJ THROUGH Course$=Course$ TO (neither the initial ("Teacher") nor the final ("Course") entity type is included).

If the relational expression is neither a projection nor a selection then the arc is a *join-arc*. In general, arcs linking non-terminal nodes are join-arcs.

The *reduction* of the graph to an equivalent relational expression is recorded in the MS if the graph corresponds to a DEFINE statement or used immediately (evaluated) if the graph originated as a query. The reduction is accomplished through the following recursive algorithm, starting at the root node:

1) Reduce all tail nodes connected through join-arcs, of which this node is the head. This step causes the algorithm to reduce recursively the graph by starting at nodes having a terminal-node as a tail-node, leaving itself as a terminal-node.

2) Build the intermediate relational expression I-REL by appending the expressions of all non-join arcs with SELECT operations to the node's entity type name. If none such node is found, I-REL contains the node's entity type name only. The objective of this step is to reduce the cardinality of the relations involved by inserting SELECT operations before JOINs or OUTERJOINs.

3) Append to I-REL the expressions resulting from repeatedly outer joining all the relational expression that resulted from the reduction of the join-arcs in step 1. The "partial" relational expressions are completed by including the entity types of the participating nodes.

4) Append to I-REL a PROJECT operation containing the name of the functions of the rest of the (terminal) nodes, and its own E-attribute. Return this as the result. This step is not strictly necessary but it may help decrease the volumes of data needed in further operations. This may reduce the amount of data traffic in a distributed environment where different portions of the graph are processed at different sites.

5) Append the PRINT-list to the final relational expression, indicating also the name of the entity type of each printed property (this is to avoid name conflicts).

Following the example, the algorithm would start at node "Teacher" and proceed immediately to node "Course" (*b* is the only join-arc). At this point, neither join-arcs nor arcs with SELECT operations are found. I-REL is assigned a PROJECT operation containing "title" and the surrogate attribute of "Course": I-REL := PROJECTED TO title,Course$. This is returned as the result.

Back to node "Teacher" and step 2, I-REL is assigned the following expression: I-REL := Teacher SELECTED ON name = "Carmen".

In step 3, I-REL is appended the relational expression resulting from outerjoining "Teacher" and "Course" and join-arc *b*. In step 4, a project operation is appended and finally, in step 5, the Project list is appended. The result is the following:

```
I-REL := ((Teacher SELECTED ON name = "Carmen")
 UOJ Teacher$ = Teacher$ TO course-Teacher
  UOJ THROUGH Course$ = Course$ TO
    (Course PROJECTED TO title,Course$))
      PROJECTED TO Course.title
```

The above procedure is a particular case of the more general situation illustrated in the following example:

```
For EACH Teacher SUCH THAT
   SOME Student HAS
      name(Student) = name(Teacher)
   PRINT name(Teacher)
```

In this case, the entity types are linked through a predicate and not because of being the argument of one another. This situation generates two sub-graphs, connected through the predicate expression, which represents the arguments of the JOIN operation. We add the following step 4.1 to the above algorithm:

4.1) Each subgraph is reduced as a separate graph and their final expressions are joined through the predicate expressions.

In fact, a query may have to be decomposed in more than one graph. Consider the following example:

```
FOR EACH Teacher
   FOR EACH Course
      PRINT name(Teacher), title(Course)
```

In this case, there is no connection between the entities of the query and it may therefore be decomposed into two separate queries i.e two graphs, each of them analysed separately:


```
query 1:  Teacher  PROJECTED TO name
query 2:  Course   PROJECTED TO title
```

4.6.4 EFDM I/O and Updates

Bulk loading of an EFDM database from external files as well as update operations has been investigated in [Kulkarni 83]. EFDM includes various syntactic constructs for updates of both data and meta-data as well as bulk-load operations. We illustrate only a few of the update operations through some examples:

a) Create a new "Teacher" entity:

```
FOR A NEW t IN Teacher
   LET name(t) = "Erica"
   LET salary (t) = 12000
```

The above insertion can be translated into an update operation to the ICS through the IDML operation "add tuple" (see chapter 6):

```
add tuple ("Teacher", {Erica,12000})
```

The underlying system generates the corresponding surrogates and updates the E and P-relations according to the integrity rules.

b) Update an immediate property:

```
FOR THE t IN Teacher SUCH THAT
   name(t) = "Mariana"
   LET salary(t) = 15000
```

This update may also translated into IDML operations using the "replace tuple" operation.

```
c) Create a new association: add the course
"Logic" to the philosophy Student "Josefina"

FOR THE s IN Student SUCH THAT
  name(s) = "Josefina" AND
  faculty(s) = "Philosophy"
  INCLUDE course(s) = c IN
    { THE c1 IN Course SUCH THAT
        title(c1) = "Logic" }
```

This association may be translated into a series of IDML statements, first finding the surrogate of the entities "Student" and "Course" referred to in the request, then creating a new associative entity (producing a new surrogate for the "course-Student$" attribute) and finally adding a tuple to the associative relation "course-Student" with the above information.

Since an EFDM query is translated into a relational algebra expression, it can be analysed by the common Syntax Analyser and evaluated by the common Query Evaluator. The result of the query is therefore a relation which can be displayed as any other relation by the common procedures. The same case applies for EFDM queries generating more than one relational expression.

4.7 Deriving an EFDM schema from the ICS

The question of whether a User Conceptual Schema is to be created automatically by the system (and if so, how?) or by the user somehow directing the system on what to do (or a mixture of both) is analysed in this section.

Mapping from a relational schema to an EFDM schema may be done according to the following rules:

RELATIONAL		FUNCTIONAL
relation	-->	entity type
attribute	-->	function over the entity type (immediate property)
tuple	-->	entity

In the case of an attribute, the range of the function is the domain of the attribute. If the attribute is a primary key then the function must be total and one-to-one. If it is a candidate key then the function can be partial but it must still be one-to-one.

The next sections describe the methods used for deriving an EFDM User Conceptual Schema (UCS) from a *basic* ICS and from an *advanced* ICS.

4.7.1 Deriving an EFDM UCS from a basic ICS

We assume that a database has been created through a basic relational query language such as TABLES and after working with it, a user wants to access the data through EFDM. To do this, the user or the system will have to create an equivalent EFDM UCS out of the existing basic ICS. Consider the relational database shown in figure 4-1, created by a basic system i.e. all relations are of type kernel. The description of the database is shown in figure 4-2.

There is more than one possible EFDM UCS schema that could be produced from this database, depending on the artificiality of the linking relations "Organise", "Attend", "Workfor" and "Belong". By using a basic system, the user was forced to create them to link the relations "Teacher", "Course", "Department" and "Student" (a basic system does not know about associative types).

In RM/T those linking relations would have been created as associative entity types, leaving to the system the task of linking entities through appropriate surrogates (but directed externally through a command such as ASSOCIATE Teacher WITH Course SUCH THAT Teacher.name = "Carmen" AND Course.title = "Bird Watching").

It could be argued that the user should have the choice of either continuing with the artificial entity types or using some system command to automatically re-organise the database transforming the linking relations into associative types. We adopt the "coexisting" approach based on the assumption that the existing system may be used by different users and a number of applications, which would prefer to keep the database organization as it is. The system therefore automatically produces the set of functions corresponding to the existing relations in the basic system, and their corresponding immediate properties.

The "advanced" user may define a set of derived functions (using DEFINE) providing a more suitable link between entity types, such as:

```
DEFINE Course(Teacher) ->> Course SUCH THAT
  SOME Organise HAS
    t#(Teacher) = t#(Organise) AND
  SOME Course HAS
    c#(Course) = c#(Organise)
```

The next section provides a method for deriving an EFDM UCS from the ICS. The case of deriving it from an ICS created through a basic system, such as the one discussed in this section, is a particular case of the general method.

4.7.2 Deriving an EFDM UCS from the ICS

Each of the ICS structures has an isomorphic UCS function definition. The method of constructing it is essentially the inverse of declaring functions in EFDM and consists of analysing each of the ICS graph relations.

1. Property Graph relation (PG). Each tuple in the PG relation defines a function. The

function name is derived from the P-relation attribute value, the function argument is taken from the E-relation attribute value and the result type comes from the Catalog relation "attributes" which contains the attribute type (it has the attributes "rel-name", "att-name "and "att-type").

The corresponding functions are created through the following procedure:

a) Join the PG and attributes relation:

```
R := (PG JOINED THROUGH P-Relation = att-name TO attributes)
PROJECTED TO E-relation, attribute-name, attribute-type
```

b) Declare a function out of each tuple in R. The function name is the attribute-name value, its argument is the E-relation value and the result type is the attribute-type value.

The PG relation shown in figure 4-4 would produce the following function declarations:

```
DECLARE  t#(Teacher)  ->  Integer
DECLARE  name(Teacher)  ->  String
etc
```

The task is simpler if compressed relations are used: each attribute, except those of type surrogate ("Surrog"), of the kernel relation defines an immediate property.

2. Generalization Graph relation (GG). Each entity type at the top of each generalization hierarchy is declared as of type "Entity". Then, each tuple in the GG relation generates a declaration of an entity type such that the "Subtype" attribute value identifies the entity name and the "Supertype" attribute value identifies the entity type.

The GG relation shown in figure 4-7 would produce the following declarations:

```
DECLARE  Person  ()  ->  Entity
DECLARE  Teacher()  ->  Person
DECLARE  Student()  ->  Person
etc
```

The top entities in each hierarchy are found by producing the set difference between the Supertype attribute and the Subtype attribute:

```
Top := (GG PROJECTED TO Supertype) - (GG PROJECTED TO Subtype)
```

3. Characteristic Graph relation (CG). Each tuple in the CG relation declares a multivalued function in the UCS. The "Sub" attribute determines the function's name and the result type and the "Sup" attribute determines the function's argument. The CG relation shown in figure 4-5 would generate the following declarations:

```
DECLARE  Job(Teacher)  ->>  Job
DECLARE  Salary-History  (Job)  ->>  Salary-History
DECLARE  Blood-Type(Blood)  ->>  Blood-Type
```

4. Association Graph relation (AG). The AG relation associates each of the immediate participants of the associative type. It is therefore possible to declare a function linking each pair of participants. The "Association" attribute declares an entity type. Since there is no way other than through integrity constraints to determine whether an association is 1:m, m:1 or

m:n, in the absence of such a constraint, the function is declared as multivalued, which is less restrictive. The AG relation shown in figure 4-6 would produce the following declarations:

```
DECLARE  Enrolment()          ->>  Entity
DECLARE  Course(Student)      ->>  Course
DECLARE  Student(Course)      ->>  Student
```

In the last case, and provided that the system can handle inverse functions, the last declaration could be rewritten as:

```
DECLARE  Student(Course)  ->>  Inverse of
         Course(Student)
```

After processing all graph relations, the system checks the Catalog relation "relations" for any relation not declared. If any is found not declared, then it is of type kernel and declares an "Entity" type. Finally, a set of meta-functions that apply the UCS objects may be also provided:

```
DECLARE  function()  ->>  Entity
DECLARE  name(function)  ->  String
DECLARE  arguments(function)  ->>  String
DECLARE  resultype(Function)  ->  function
```

4.8 Conclusions

The coexistence of external data models is complicated by the presence of models offering different levels of semantics. This is mainly due to the higher level of abstractions and implicit constraints introduced by the advanced systems, which are not easily reflected in basic systems. It is expected that, provided a choice of data models exist, a database designer will choose the one offering the modeling tools required for his application, and will not change this later on. In other words, a database is not likely to be modeled with a basic model and afterwards upgraded through reorganizations towards an advanced model. What is expected is that given a basic system, a user may want to use the abstraction capabilities of an advanced system for its processing requirements (e.g. DEFINED functions) without introducing advanced structures. On the other hand, given an advanced system, some users may prefer a simplified (perhaps more familiar), although restricted view of the application being modeled and access it through a basic system.

We have investigated some of the concepts needed for a "good" internal model and concluded that RM/T is capable of supporting them. On the other hand, RM/T offers a "semantically open-ended" approach to data modeling in that new abstractions may be introduced by defining new graph relations and integrity rules governing the insertion and deletion of its tuples.

Although the methods shown address only a subset of the capabilities offered by RM/T and EFDM, they establish a framework in which data models with different levels of semantics may coexist. Therefore, these methods are regarded only as an initial attempt in mapping the functional model, FQL and EFDM in particular, to the RM/T model. There is clearly a need for more research in the area, especially regarding the more sophisticated EFDM capabilities such as inverse functions, transitive closures and view and constraint handling. More research

is also needed in the mapping of other RM/T structures, the use of the new operations and the treatment of null values, all of which should be supported by an implementation.

The implications are however, that RM/T emerges not only as a powerful database design tool, but that it can be effectively used to support the semantics offered by widely different models other than the relational. The RM/T model is therefore considered a suitable candidate for the internal data model, capable of supporting the coexistence of a multi-model external world.

Chapter 5
PS-algol

5.1 Introduction

It is claimed in this thesis that an adequate database architecture can support the implementation of a database environment in which new user interfaces can be quickly and economically added. However, the most important tool in constructing the architecture's components and new interfaces is the programming language. Most of the software for this research was written in PS-algol which is the result of incorporating into the language S-algol [Morrison 81, Cole 82]the ideas developed by Atkinson [Atkinson 78a, Atkinson 78b, Atkinson 79, Atkinson 81b, Atkinson 83b]. The availability of a language such as this, with persistence as an orthogonal property of data of any type, makes it easier to implement a database environment within reasonable time and with few resources.

Despite being a prototype language that is still under development and test, it is remarkably robust and complete. The designer's emphasis has been mainly on functionality rather than efficiency and given the peformance achieved so far for relatively small volumes of data, there is every reason for believing that, after devoting some effort to optimization, PS-algol will emerge as a very competitive language for large databases. For example, the code is presently being interpreted.

The following section explains the reasons for choosing PS-algol instead of one of the other locally available languages Imp [Robertson 80] and Pascal. Next, some of the outstanding features of PS-algol as well as its limitations are presented.

5.2 Alternative Languages

Before PS-algol became available, its underlying transaction management system, called the Chunk Manager (CMS) [Atkinson 83a] was used as the physical store manager. The CMS is offered as a library of external procedures that can be linked to Imp or Pascal programs. It stores, retrieves and administers chunks of arbitrary and varying size on disc. It also includes an associative lookup mechanism in the form of B-tree tables that map strings to integers.

By providing a machine and operating system independent environment, the CMS facilitates

the fast prototyping of database systems. However the CMS was designed as a low level data handler to be used in any application, independent of its data model. It does not record any structural information with the data and it is therefore the application's responsibility to ensure type consistency and provide adequate structural transformations. It is the PS-algol language that goes an important step forward by managing the persistence of arbitrary data structures and types.

Despite having written a substantial amount of the software in Pascal using the CMS, I decided to switch to PS-algol. The main reasons were:

1) After implementing a few short programs in PS-algol, its potentiality for novel approaches in writing DBS software were apparent. Some experiments in searching for alternative relation structures are reported in chapter 8.

2) The first experiments demonstrated that the PS-algol run-time system was at least as fast as the Pascal/CMS version. This was partly due to the reduction in software layers necessary to reach data that had already been imported to the heap. At that time I only tested very small volumes of data e.g. 100 tuples in a relation. Later experiments with up to 10,000 tuples showed that the performance became increasingly non-linear as more data was used. This situation was significantly improved through better hashing algorithms as well as other techniques, which have pushed the linearity break-point beyond the needs of small applications.

3) Those first experiments gave a factor of about three in favour of PS-algol in both coding time and program length. As programs became larger, it was necessary and desirable to decompose them in modules. Because good modularisation aids were not initially provided, coupled with poor runtime diagnostics, the coding factor was reduced to about only two. Again, this problem has been overcome recently and the 3:1 ratio will probably be surpassed.

5.3 Experience with PS-algol

After coding a few thousand lines in PS-algol and experiencing some important improvements in processing speed and robustness, its distinguishing features as well as some drawbacks can be highlighted. This is done with respect to its ability in assisting in the implementation of DBS experiments, with emphasis on fast prototyping, reusability of components, robustness , reliability and diagnostic aids.

5.3.1 The Good Features

It has been reported that typically around 30% of the total code in any program is devoted to transferring data to and from files. This amounts to more programming effort, slowing down the software production turnaround as well as its processing time. It often requires the conversion of any structure created with the data to and from file formats with the consequent loss in type protection [Atkinson 83c]. This major disadvantage of conventional programming languages is avoided in PS-algol through the provision of persistence for any data, regardless

of its type. The simplicity in achieving it and the uniformity in providing it is obviously the best feature of this language.

PS-algol programs handle persistent and transient data in the same way and without exceptions e.g. a procedure can be called with its parameter being transient or persistent data [Atkinson 83c].

The logical to physical mapping of objects, and the physical retrieval and placing of them is an important and time consuming part of any database system that must be properly designed to get it done efficiently and reliably. It requires extra expertise from the programmers in that they must be well acquainted with the underlying hardware characteristics. PS-algol relieves the programmer from that burden making it therefore simpler to build incrementally a database environment. Leaving the physical object management to the language run time system is a major cause for reducing code lines and programming effort. For example, in the Edquse system, from its 6155 imp code lines, spread over 10 modules, 2 modules with 735 lines (12%) deal exclusively with buffer and file handling. Those modules are the most difficult to maintain and have proved to be the principal source of errors.

Simplicity and power, two fundamental design principles in its ancestor language S-Algol, [Morrison 81], are other good reasons for having smaller and more readable programs.

Data type completeness, with no exceptions to remember makes a manual a scarcely needed document. Particularly useful are the locality of declarations with initial values determining the object type, string handling facilities, flexible and varied I/O functions, associative tables (B-trees), dynamic vectors and untyped pointers.

All these advantages have a positive impact on the speed and quality with which one can produce prototypes and test ideas. The impression is that one develops solutions assisted by the language and not despite it.

5.3.2 Some Drawbacks

All of the following problems were apparent while writing the software. However they have been gradually solved and therefore do not constitute a valid criticism of PS-algol as it will be seen in public versions.

Isolation from other software: one of the basic ideas behind rapid prototyping is the reusability of software components. PS-algol is presently an isolated language i.e. there are no means for communicating with software written in other programming languages. This was particularly frustrating in my case because I had already implemented an important part of the software, especially utility routines that did not need redesign (e.g. type conversion, i/o, etc.).

Unreliability and poor run time diagnostics were among the major reasons for lower productivity. When logic errors occur while experimenting, it is expected to find them in the experiment, not in the tools implementing it. However, this situation has significantly improved and surely the next generation of users will find it easier.

Modularisation facilities are very limited. The only use of modules is to physically partition the code in separate files so that the editor and compiler take less time. But since procedures in modules don't share declarations much coding and maintenance is required to keep say, 8 modules consistent especially when experimenting with data structures. Run time diagnostics in modules refer only to line numbers relative to the start of a procedure, whose position is not revealed.

In addition to improving the previous mentioned drawbacks, a few desirable additions are briefly mentioned: An *include* facility to avoid replication of data declarations, particularly in modules. Multitype tables: at present, only strings are allowed as keys in the table mechanism (B-trees). Procedures as first class objects: presently, procedures can be passed as parameters but not returned as results of procedures. Richer type system: reals, enumeration types, sets, etc. Re-engineering of modules. Better diagnostic aids, particularly in modules. Dynamic update of structure classes (very useful when experimenting with structures). Transactions as a mechanism for implementing experiments and backing up if necessary. Concurrency handling.

5.4 PS-algol Examples

The following two programs may illustrate some of the facilities of PS-algol.

The first program uses the PS-algol tables facilities to store some information about persons (manager and salary) in a structure. The table key is the person's name.

```
! Declare structure of class "person"
structure person (string manager; int salary)

let relation := table  ! create new table
let name     := ""

let finished := false
while ~ finished do
begin
  let p := person ("",0)
  write " Name    : "; name          := reads
  write " Manager: "; p (manager) := reads
  write " Salary : "; p (salary)  := readi
  write "'n More persons (y/n) ? "
  finished := read ~= "y"
  enter (name,relation,p)
end
! structures accessible from the root table
! are made persistent
enter ("Relation",root.table,relation)
commit
```

The next program asks for the name of person and writes the name of his manager, his manager's manager, etc.

```
structure person (string manager; int salary)
let relation := lookup ("Relation",root.table)

procedure print.manager (string name)
!-----------------------------------
begin
   let p = lookup (name,relation)
   if  p = nil
     then write "'nManager of ",name," not found"
     else
     begin
        write "'nManager of ",name,":",p(manager)
        print.manager (p(manager))
     end
end

write " Name of person: "
print.manager (reads)
```

5.5 Conclusions

In conclusion, the provision of PS-algol makes it possible to minimise the requirements for anyone intending to implement a database environment along the lines of this thesis. That is, the *only* requirement for an implementor is to be reasonably proficient in a programming language with persistence as an orthogonal property of data of any type.

This is at least an order of magnitude less than the normal expertise required e.g. operating systems, disc hardware, communications, etc. and an important step toward making database technology an affordable technique for a wider population.

Chapter 6
A Database Architecture

6.1 Introduction

This chapter presents in detail the set of components that constitutes the proposed database architecture.

In order to justify the existence of each component, and to be consistent with the evolutionary approach, the incremental design of a hypothetical system, moulding systematically a software environment centred on a database will be sketched. The total database task is divided into parts, encapsulating in each part the implementation of a database function that is first defined on a *minimal functionality* basis for a prototype system and then gradually extended. This is essentially a process of abstraction as defined by Hoare and consisting of four stages: the decision to concentrate on shared properties, the definition of names used as the communication means, the manipulation rules and the axiomatisation of the properties [Dahl 72].

A similar database environment has been implemented by the author in order to demonstrate the feasibility of it. Its user interfaces are presented in the next chapter.

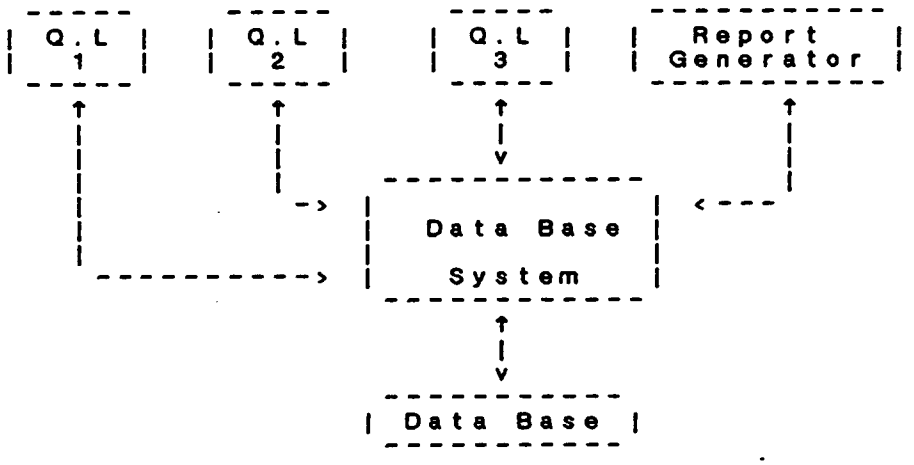
From the four disciplines relating to the prevention of errors in a database environment i.e security, consistency, reliability and integrity, we address only the latter in some detail. *Security* deals with preventing users from accessing or modifying the data in unauthorized ways. *Consistency* is the discipline of preventing errors which may arise due to the interaction of two or more processes operating concurrently on shared data. *Reliability* deals with prevention of errors due to the malfunctioning of system hardware or software. These issues, although very important, are expected to be handled, at least in part, by the underlying data manager in the language used to implement this research: PS-algol (but see [Date 83] or [Ullman 82] for tutorials on them). Another issue, distribution, is only briefly addressed.

6.2 The Target Environment

The example environment will contain a choice of query languages to meet the needs of users with different processing requirements and levels of competence, as well as other packages such as a report generator, as shown in figure 6-1.

Other interfaces should be added later at minimum cost by re-using existing software components wherever possible. This will avoid the duplication of similar software but will require flexible and general purpose components with good documentation. Components can be regarded as specialised "engines" that are parts of an abstract "machine".

Figure 6-1: Database Environment



The shareability of components will require the identification of common database tasks that are amenable to standardization. As suggested in the chapter on software engineering, this can be achieved by providing an internal conceptual shema (ICS), a common internal query language (IQL) and a common pool of data manipulation routines. All of them can be provided on an incremental basis. For example, the IQL may be started with the three "basic" relational operations project, select and join, followed at a later stage by set operations, building up gradually a fully relational system. A similar approach can be defined for the data manipulation routines and the ICS structures.

Having decided on the internal data model, which in this case is the relational model, the first user interface and the intial set of components can be designed as a prototype.

6.3 A Minimal Environment

Another advantage of implementing a database architecture as a set of components is to make it possible to implement these components concurrently. The ability to test the software and its coupling to other components while writing it, requires the definition of different

priorities in functionality. Some minimal set of functions that constitute a rudimentary but working system can be rapidly implemented, tested and then systematically modified and expanded. The first versions of a system can be regarded as prototype systems. Some of the many advantages of prototypes are outlined in chapter 3.

The hypothetical system will have at first one simple query language (Q.L.1). After some experience with it, the prototype will be expanded to meet the initial objectives. Q.L.1 is intended for non-expert users and will require only a few operators. It will help users familiarise with the system and encourage them to provide early feedback to designers in order to tune the final system towards meeting more effectively user requirements and preferences. It will also permit designers to test different implementation strategies.

To achieve fast prototyping of the user interfaces it is desirable to design them under the principle of *minimum responsibility*, that is, they should be relieved from as much processing and decision making as possible that may distract them from their two main tasks of presenting information to users and collecting their requests. The communication with the database and the processing of requests is facilitated by providing components that accomplish each of the required tasks. For example, processing a query or ensuring the proper installation of updates should be done by specialised system components.

The principle of minimum responsibility requires though, good communications between user interfaces and the internal components. That is, there is an agreement on internal standards that are understood by the system and to which the user interfaces translate their requests and read the results.

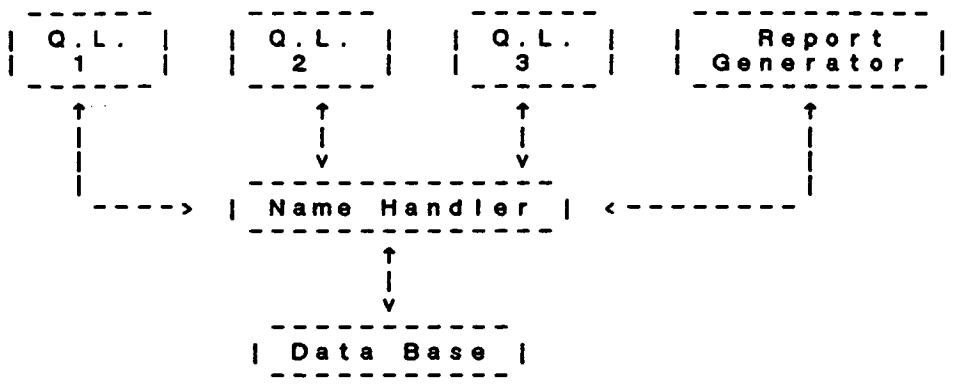
To facilitate this communication it is desirable that user interfaces refer to objects (relations and attributes) by name instead of by system provided identifiers. This will also permit the underlying system to change the identifiers if necessary (e.g. restructuring a relation) and also allow users to define different name spaces for the same objects. Thus, a first component can be defined as a Name Handler.

6.4 The Name Handler

The Name Handler (NH) will serve as the central node of the system. It receives all messages from the user interfaces redirecting them to the proper components as specified in the message. In that process, the NH performs a minimal compilation in translating names to internal identifiers. The existence of the NH isolates internal components from user interfaces, simplifying the internal references to objects by the use of internal identifiers and protecting the database integrity by ensuring the proper establishment of data paths. For example, attempts to modify the database by the users may be first checked for consistency and rights in a special purpose Update Handler, as directed by the Name Handler.

User interface components may regard the NH as the sole database component, providing all the required services. No other component may be visible to them. The database, as seen from the user interface components consists of the elements shown in figure 6-2. However, the possibility of having some user interfaces such as a performance analyser or data restructurer accessing the system at other levels, is retained.

Figure 6–2: User Interface View of the Database



For the Name Handler to translate names into internal identifiers and to ensure that the objects referred to exist, it must have access to the Internal Conceptual Schema (ICS) structures. The ICS consists of a set of relations and is therefore maintained and accessed through the same data manipulation routines as any other database relation. This uniform access to both data and meta–data reduces the total cost of software and simplifies its development and maintenance by avoiding the definition of a dedicated data description language.

From the many relations that can exist in the ICS, two of them appear as fundamental and may be considered as the "minimal" structures of any ICS. These two relations are the **relations** and the **attributes** relations containing information about all the existing relations and attributes in the database. The attributes of each of these relations may vary and can be defined initially under the principle of "minimality" and later extended as needed.

The **relations** relation should contain at least the name for each of the relations in the database (one tuple per relation). Other attributes may indicate sort order, protection, degree, cardinality, etc.

The **attributes** relation has one tuple for each attribute in the database. The name of the attribute, the name of the relation owning the attribute and the attribute type or domain name is required. If a domain name is chosen, then another relation defining it should also be provided. Other attributes of this relation may indicate default values, constraints, protection, etc.

What information should be added in the form of new attributes to these relations or as new relations in the ICS is the designer's decision.

6.5 The Logical Storage Handler

The storage and retrieval of objects can be identified as a specialised function and therefore left to a Storage Handler component. Given that the internal model is relational, it will be necessary to create, delete and manipulate objects of the model such as relations, attributes and tuples.

It is supposed that the designers use a language such as PS-Algol requiring therefore no Physical Storage Handler component. Otherwise, a Physical Storage Handler manipulating objects on buffers and possibly a File Handler communicating with the operating system's I/O functions is necessary. The lowest layer of the example system is, hence, the Logical Storage Handler (LSH). It is in charge of creating and maintaining relational structures as commanded by the Name Handler. The encapsulation of all object manipulation in a dedicated component permits the experimentation with different relational structures to meet an adequate compromise between space usage and processing efficiency. Such an exercise is presented in chapter 8.

It may also create, on demand, indexes on relations. The initial functionality required from the LSH is to permit access to individual relations, attributes and tuples. The first functions to be implemented may be those constructing objects such as *create relation*, *create attribute*, *add tuple* and of course, *get tuple*. Destructive operations such as *delete relation*, *delete attribute* and *delete tuple*, etc. can be added on a second iteration.

6.6 The Common Query Evaluator

Given a Name Handler and a Logical Storage Handler, the only component that may be required to start a user interface is a Query Evaluator. It may be first restricted to simple queries and to limited and unchecked update capabilities.

The provision of a common Query Evaluator (QE) for a multi-language system requires the definition of a common Intermediate Query Language (IQL) as discussed earlier. The IQL should be amenable to computer processing and it is therefore not required for it to be particularly user-friendly. With the internal data model being relational, it appears natural to define a relational language, although other alternatives (e.g. FQL [Buneman 82]) may also be tried. In the database environment implemented for this research, the IQL in the form of a relational algebra was defined and implemented similar to the proposals by Atkinson for an internal query language for the distributed system Proteus [Atkinson 82]. It is briefly presented in the next section.

6.6.1 The Intermediate Query Language (IQL)

The IQL is proposed as a candidate internal database standard. Its form has been chosen to show the following characteristics:

1. A simple subset of it can be implemented easily.

2. It is extensible to meet new processing needs.

3. It is amenable to transformation and therefore an optimizer is feasible.

4. It is machine independent and therefore queries can be transported to any installation implementing it.

The IQL consists of a set of one and zero address instructions intended for a hypothetical stack machine. Each instruction is denoted by a mnemonic word that may also indicate the type of the instruction. Initially the types are one of boolean (b), integer (i), string (s) or relation (r). Other possible types are float (f), time (t), date (d), money (m), etc. Each instruction may be uniquely labelled by a word preceded by a dollar sign. There is a reserved label $start which defines where to start evaluating the query.

As an example, consider a database having a relation "Person" with the attributes "Name", "Sex" and "Address". The query "List the name and address of all females" can be expressed in IQL as in figure 6-3. In most of the instruction mnemonics, it is the first letter (e.g. r,s,b) that denotes the type of the instruction.

Figure 6-3: Intermediate Query Language: Query 1

```
$start rload Person ! load relation id onto stack
       rselect S    ! select tuples using label S
       rproject P   ! project attributes using label P
       rresult      ! end of main query body

$S     sload "f"    ! load string "f" onto stack
       satt Sex     ! load (string) attribute value
       sequal       ! compare top stack items
       bresult      ! result is a boolean on stack

$P     satt Name    ! load string att. Name on stack
       satt Address ! load string att. Address on stack
       end          ! end of projection list
```

The example in figure 6-3 shows the anatomy of an IQL request. Basically, it consists of a main body indicating the relational operations involved in the query, and a label associated with each of them that define how to process the operation.

The evaluation of a query can be done in a serialized mode as follows: the processing of each relational operation is controlled by a dedicated *relational engine*, which takes a set of instructions and one or two relations as input and delivers one relation as output. Each relational engine invokes the stack machine with one tuple of its input relation at a time (or one tuple of each input relation in the case of binary operations), and the instructions under its label. The stack machine executes the instructions under the label, using attribute values from the tuple(s) if necessary and places the result on the stack, which is consulted by the relational engine to decide whether the tuple(s) is (are) included in the result relation.

The select operation pops the boolean value off the stack, resulting from the processing of a tuple by the stack machine, to decide whether to include that tuple in the result relation or

not. In the project operation case, the attributes to be included in the result relation are collected from the stack.

The above process is only a simplified (or perhaps the initial) method of evaluating a query. A more sophisticated engine would consider, for example, the use of indexes and the sort order of tuples of the input relation(s) to speed up the evaluation time.

A different strategy is to use a pipelined method of evaluation [Todd 76]. The use of *data pipelines* enables the query evaluator to process relational expressions without creating intermediate results. A query is parsed into a tree consisting of nodes and *pipes* connecting the nodes. Each node represents a relational operation and controls the data flow in its own section of the pipe by releasing tuples to operations higher in the expression tree and demanding tuples from lower nodes. The *leaf* or *terminal* nodes of the tree correspond to stored relations which are traversed a tuple at a time, passing the tuples through the tree structure. Tuples at non-terminal nodes are generated by processing the input tuples according to the associated relational operation in the node. Once a tuple reaches the root node, it is stored as part of the query result and/or presented to the user. Pipelined evaluation can be implemented through co-routines [King 81] but requires a flexible buffer policy to avoid performance degradation due to high page faulting in complex queries where many relations are traversed in a tuple at a time mode.

The text of an IQL query consists of a sequence of statements conventionally one to a line. Each statement has the form:

```
<label> <instruction name> <parameter>

the <label> is either empty or "$"<word>

<word> is a sequence of letters, digits or
character underline (_), starting with a letter.
words may be delimited by a non-word character.

<instruction name> is one of the words defined
in [Atkinson 82] (some are listed below).

<parameter> is either a label word or a <constant>

<constant> is an integer, string, boolean or an
attribute name.
```

The following rules apply:

1. Parameters must have the expected type as specified in the instruction name.

2. A label may only be defined once.

3. A label which is used must also be defined.

There are essentially two levels of language, the relational and the scalar language. The relational language defines operations on relations. Some of these are:

- rload : this loads the relation reference denoted in the parameter onto the stack.

- rselect : this takes one parameter, the label of the encoding of a select expression. The action is to apply the selection operation specified by the expression under the label to tuples from the relation at the top of the stack, and replace the top of the stack with a reference to the relation containing the selected tuples.

- rproj : this takes one parameter, the label of an encoding of the list of attributes onto which the relation denoted at the top of the stack is to be projected. A reference to the projected relation replaces the input relation on the stack.

- rjoin : this takes one parameter, the label of an encoding of a list of triplets: two attributes participating in the join and a comparison operator that defines the type of join. A join may be defined on more than one pair of attributes. The comparison operator is normally equality but it can be any one defined for the attribute types (e.g. inequality, greater, etc. or "earlier" and "later" for attributes of type time)

Similarly, other relational operations are defined for cartesian product, set operations, sort, outer-join, etc.

The scalar language includes operations for each of the scalar types defined. Some of them, based again on the proposals by Atkinson, are:

- iload : this takes one parameter, an integer constant that is loaded onto the stack.

- iatt : this takes one parameter, the reference to an attribute whose value on the current tuple is loaded onto the stack.

- iequal, inequal, igt, ige, ilt, ile : these parameterless instructions compare the top two items of the stack and put the boolean value resulting from their comparison on the stack. The two integers are removed from the stack.

- iplus, idiff, imult, idiv, irem : these take no parameters and combine the top two items of the stack according to the specified arithmetic operation, and leave the resulting item on the stack.

- ineg : this inverts the sign of the top item on the stack.

Similarly, a number of operations for each of the other types (string, boolean, etc.) as well as data type conversion operations (e.g. integer to string) are defined.

Some string operations, similar to those for integers are: sload, satt, sequal, snequal, sgt, slt and sle. Other string operations permit the concatenation of strings, substring comparison, string comparison ignoring case (upper or lower case characters are treated identically), obtaining the length of the string, testing of string inclusion, etc.

Another set of instructions are used for control, to create new attributes (e.g. in a grouping operation) and to define group functions.

The IQL, as presently implemented, is not capable of representing properly function definitions. Instead, the IQL code includes some *grouping* functions such as sum, count, max, min, all, any and average that are also recognized and processed by the common Query Evaluator. This is clearly an area for further extensions.

6.6.2 The IQL Stack Machine

The abstract stack machine for the IQL has been implemented in a separate component, the common Query Evaluator (QE), that is tightly coupled with the Storage Handler (LSH). It "knows" the structures used by it and takes advantage of it to speed up processing.

The IQL is not passed to the QE as a sequence of instructions but as a tree where each node contains a relational operation (including rload), and its parameters, which are the instructions under the label of the operation as shown in the example in figure 6-3.

The conversion of IQL from tree form to sequence and viceversa is a trivial procedure. The sequential form is more suitable for transporting the query (e.g. to another site) or to print it for checking. The tree form is more suitable for processing, since each node is a structure that may contain additional information for the Query Evaluator such as the use of indexes, sorting specifications (e.g. a project operation may delete duplicate tuples after sorting its output relation but the sort attribute may be indicated in the node so that it is useful for a next select or join operation). A tree also may be subject to pipelined evaluation. In addition, the tree is more easily modified by an optimizer that may rearrange nodes to get more efficient processing (e.g. in general, select operations should be done before joins, unless the database has indices for the join attributes but not for the select attributes [Blasgen 76] and, of course, the query evaluator takes advantage of such information).

The example in figure 6-4 shows another IQL query, the resulting tree structure is shown in figure 6-5. It corresponds to the query "list the names of the teachers organising courses in the science faculty". The relations *Teacher* and *Organise* are linked through the common attribute *p#*. The relations *Organise* and *Course* are linked through the attribute *c#*.

The stack machine in the Query Evaluator, as implemented for this research, uses separate stacks for each data type. The nodes are recursively processed starting from the root. Each leaf node corresponds to an *rload* operation returning a relation identifier. Every non-leaf node invokes first the processing of its (at most two) underlying branches, it then creates an empty result relation according to the attribute specification included in the node structure and then proceeds to evaluate its associated list of IQL instructions, appending qualifying tuples to the result relation which is returned (loaded on top of the relation id stack).

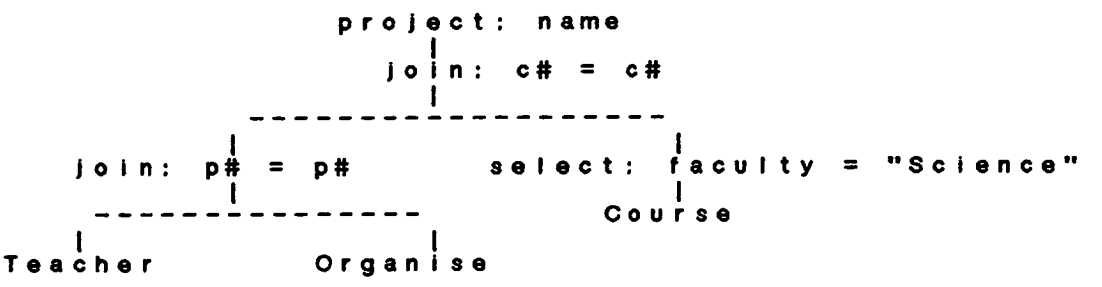
Figure 6-4: Intermediate Query Language: Query 2

```
Relations:
   Teacher   : p#, name, address, sex
   Organise  : p#, c#
   Course    : c#, title, faculty

Query: "list the names of teachers organising
        courses in the Science faculty".

Query in some relational algebra:

R1 := Teacher JOINED THROUGH p# = p# TO Organise
R2 := Course SELECTED ON faculty = "Science"
R3 := R1 JOINED THROUGH c# = c# to R2
RESULT := R3 PROJECTED TO name

Query in IQL:

$start  rload     Teacher    ! load rel.id onto stack
        rload     Organise   ! load rel.id onto stack
        rjoin     R1         ! join top stack rels
                             ! leaving result on stack
        rload     Course     ! load rel. onto stack
        rselect   R2         ! select top stack rel.
                             ! leaving result on stack
        rjoin     R3         ! join rels on top of stack
                             ! leaving result on stack
        rproject  RESULT     ! project top stack rel.
                             ! leaving result on stack
        rresult              ! result of query is rel.
                             ! id on top of stack. Stop.

$R1  pairhead  p#     ! load p# att of top stack rel
     pairtail  p#     ! load p# att of top stack-1 rel
     iequal           ! test integers for equality
     bresult          ! bool result on stack. Return

$R2  satt  faculty     ! load att value onto stack
     sload "Science"   ! load string onto stack
     sequal            ! test strings for equality
     bresult           ! bool result on stack. Return

$R3  pairhead  c#     ! load c# of top of stack rel.
     pairtail  c#     ! load c# of top of stack-1 rel.
     iequal           ! test integers for equality
     bresult          ! bool result on stack. Return

$RESULT satt name     ! load att value onto stack
        end           ! return
```

6.7 A First Prototype

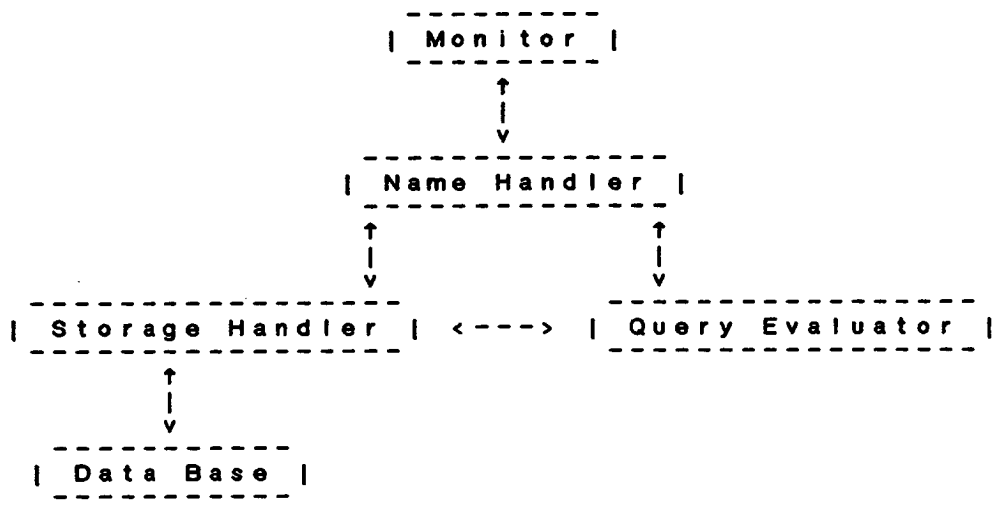
The existence of the IQL suggests the implementation of a low level monitor to test the initial versions of the NH, LSH and QE directly, prior to the implementation of Q.L.1, or as part of it. That is, a prototype for a prototype. One of the advantages of this technique is to postpone or separate the task of designing the user interface in order to allow immediate testing of the components. A simple interactive command processor and query constructor, communicating with the Name Handler can be provided. Queries are posed directly in the internal form, identifying objects by name and leaving to the Name Handler the task of

Figure 6-5: Tree for Query 2



translating them to internal identifiers and producing the tree structure. At this stage the architecture of the system appears as in figure 6-6.

Figure 6-6: Prototype Architecture



The "minimal" functions required by each of the components defined so far are:

Name Handler:

open database (database name) -> database id
if the database is new, this function constructs the "minimal" ICS: the "relations" and "attributes" relation.

add tuple (relation name, tuple) -> tuple id

get tuple (relation name, tuple id, which) -> tuple id
which : first, next, last or previous

evaluate query (query name) -> relation id

Logical Storage Handler:

create relation -> relation id

add attribute (relation id, attribute type) -> attribute id

add tuple (relation id, tuple) -> tuple id

get tuple (relation id, tuple id, which)
which : first, next, last or previous

Query Evaluator

evaluate query (query tree) -> relation id
project, select and join (in that order)
working for integers and strings only at
first and then for other types, is the
easiest implementation path.

The above functions are assumed to have proper error handling which must be incorporated in each component to check that they are importing the right objects or else abort gracefully.

While the monitor is being implemented and used to test the component's functions, interfacing, error handling and the proper generation of IQL, Q.L.1 can separately begin to produce IQL queries that are first passed "manually" to the monitor and later through a procedural interface. Finally, Q.L.1 is directly connected to the Name Handler.

6.8 The Update Handler

The Update Handler (or integrity subsystem) is in charge of ensuring the database consistency by monitoring update requests to detect integrity violations. It enforces the integrity rules (constraints) defined for the database by taking the appropriate actions if an attempted violation is detected e.g. it may reject the operation, report and record the violation or initiate automatic corrective actions [Eswaran 75]. It may also reject the definition of a new rule if this rule is violated by the current state of the database.

If an update is made on a meta-data relation (e.g. adding a tuple to the "attributes" relation, the Update Handler is in charge of triggering the corresponding procedures as side-effects of the update (e.g. adding the new attribute to the corresponding relation). Triggers are a means of implementing the integrity rules by automatically executing special-purpose procedures whenever a "trigger" event occurs [Date 83]. It is also important that the rules are subject to queries and updates, as any other piece of information in the database.

Date and Hammer and McLeod classify integrity rules in two categories: domain integrity rules and relation integrity rules [Hammer 75, Date 83]. Domain rules describe a domain as a set of atomic data objects and relation rules specify a property or relationship that must hold on or between one or more relations or subparts of relations (tuples or attributes).

Every attribute has an underlying domain which is a subset of one of the *base* domains e.g. the set of all strings, the set of all integers, etc. Domain integrity rules are associated with the definition of the domain which either explicitly or implicitly specifies all acceptable values for the domain. Domain rules are considered in isolation i.e. independent of other attribute values in the database.

In our implementation an attribute may be associated with a predicate that defines its domain. For example, the attribute "age" of a person can be associated with the predicate (age >=0 AND age <= 150). The Update handler ensures that every new value of this attribute satisfies the predicate. For this purpose, the Update Handler maintains a "compiled" version of the predicate, in the form of a query, that is sent to the common Query Evaluator together with a tuple holding the candidate attribute value. If the stack machine produces a "true" value, the attribute is accepted. Another domain rule allowed is that of enforcing that the values of an attribute in a relation are "unique" i.e no duplicate attribute values may exist.

Relation integrity rules concern the admissibility of a given tuple as a candidate for insertion into a given relation, or the relationship between tuples of one relation and those of another. In our implementation we allow the definition of *referential* integrity rules specifying that the values of a given attribute must also exist as a value in an attribute of another (or the same) relation. For example, the values of the attribute "rel-name" in the ICS "attributes" relation must also exist as values of the "rel-name" attribute in the ICS "relations" relation.

The concentration of all integrity rules in one component, instead of scattered across several components makes these rules easier to maintain, extend and check.

6.9 The Utilities Component

During the implementation of the components, it becomes clear that many housekeeping routines can be shared among components. These routines can be incorporated in a Utilities component. Furthermore, once the second Q.L. (Q.L.2) is started, checking and reporting of user errors may also be shared.

Some of the functions that may be included in the Utilities Component are:

a) **I/O** (from/to a terminal or an external file): reading a query, reading data and ensuring that it belongs to a given type e.g. reading/checking date or time type input. Displaying help and error messages, querying results, etc. Output to an external file of a relation (in both formatted and unformatted modes). Loading a relation from an external file, etc.

b) **Data Type conversion and String handling:** our implementation stores all data as strings and therefore requires some type conversion routines: integer to string, time to string, string to integer, etc. as well as some string handling facilities such as extracting the first (next, or any) symbol of a string according to the symbol delimitation rules (e.g. spaces, newlines, tabs).

c) **Vector Handling:** much of the data (e.g. tuples) are manipulated as string and pointer vectors in our implementation. The following routines facilitate their handling:

```
extend.s.vector (*string vs; string s -> *string)
  given a vector of strings vs and a string s, it
  appends s to vs (extending the vector by one element)
extend.p.vector (*pntr vp: pntr p -> *pntr)
```

```
    as extend.s.vector for pointer vectors.
  copy.s.vector (*string a -> *string)
  copy.p.vector (*pntr p -> *pntr)
  shrink.s.vector (*string sv; int i -> *string):
    given a vector of strings sv and an integer i, it
    returns a smaller string vector without the i-th element.
  shrink.p.vector (*pntr pv; int i -> *pntr):
    as shrink.s.vector for pointer vectors.
```

d) **Functions over names**: given a name a component may need to check whether it is a base relation, a query, a view, an attribute name, etc. In the case of an attribute, it may need to know its type, constraints, default value, position, etc. Different routines handle these requests.

6.10 The Optimizer Component

As the volumes of data grow and the processing requirements become more complex, a system should include an Optimizer component that establishes the evaluation strategy of a query. The objective is to relieve the users, especially casual users, from the need to know any detail that would make a query yield faster a result. In other words, the speed with which a query is evaluated should not depend critically on the way the query is formulated [Hall 75]. An optimizer will typically calculate the cost of evaluation of a query, due to possibly different evaluating strategies, according to a cost model and select the strategy with minimal cost. The cost may be expressed in both cpu time and number of disk accesses. Jarke and Koch have surveyed many query optimization strategies [Jarke 82].

Yao, Blasgen and Eswaran propose various methods of evaluating a query based on a storage and access model of a relational database [Yao 79, Blasgen 76]. The methods include the use of indexes on join attributes, use of indexes on select attributes, sorting relations and different evaluation orders of the operations involved in the query. A number of cost equations are given that consider various storage and access path parameters. The methods are compared for various parameter values and the conclusion is that they can effectively reduce the time of query evaluation. Blasgen and Eswaran indicate a global strategy for an optimizer: first, given a query, determine the applicable methods from the available access paths. Second, eliminate any obviously bad method and those failing to pass certain simple tests. Finally, evaluate the cost estimates for the remaining methods and choose the method with minimum cost. They conclude that optimization methods should take into account existing access paths and their properties.

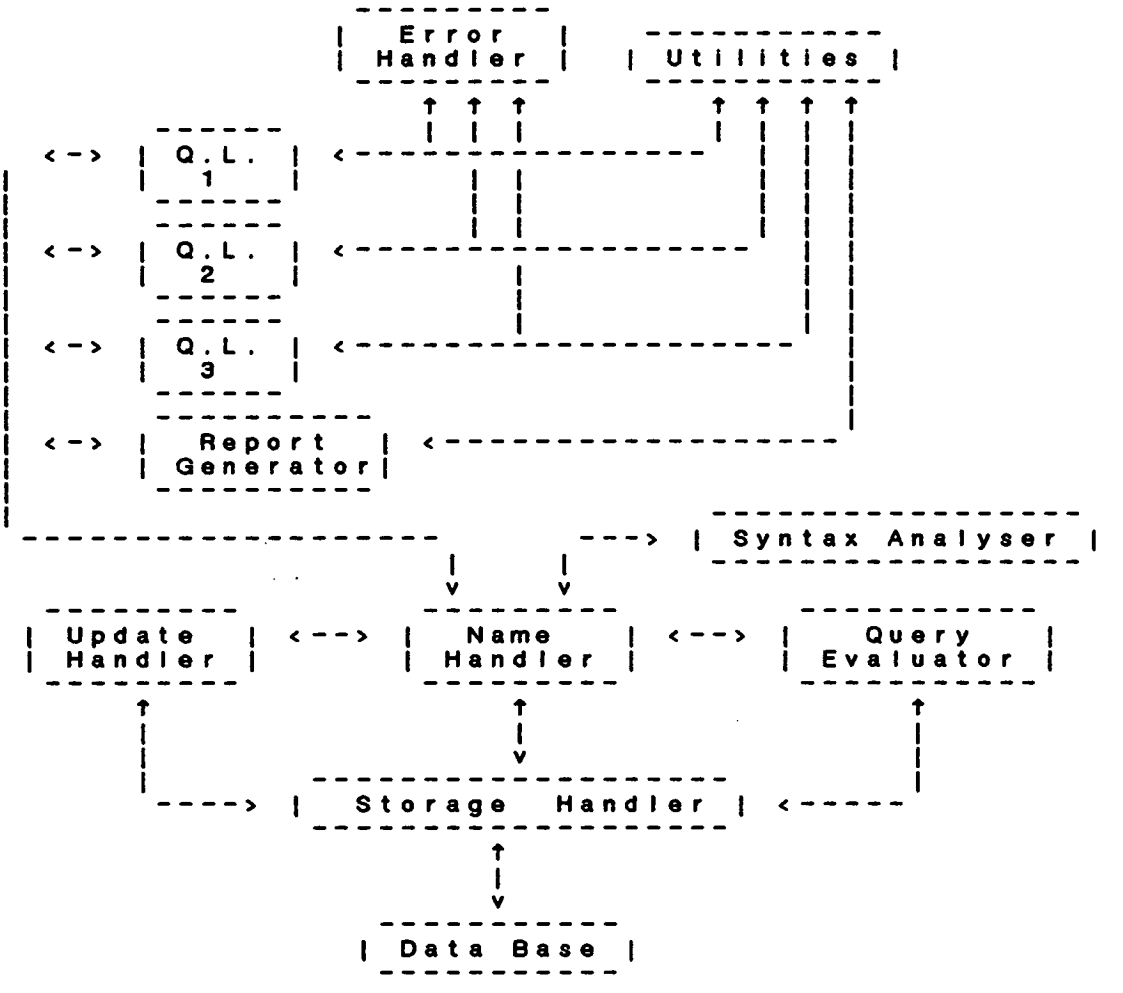
6.11 An Extended Environment

In the implemented environment reported in the next chapter, two internal query languages are used. One is a higher level relational algebra, called RAQUEL (Relational Algebra Query and Update Extensible Language), which is produced by the user interfaces. It is in fact one of the query languages offered to the users. RAQUEL queries are passed to a common Syntax Analyser component (through the Name Handler) which in turn produces the IQL tree. This second indirection, although introducing some overhead, simplifies the implementation of

user interfaces by leaving to the common Syntax Analyser the checking of most of the errors introduced by users when defining a query. It is also considerably easier to generate a query in this relational algebra than in IQL due to the higher procedurality and verbosity of IQL.

The architecture implemented for this research is shown in figure 6-7. To simplify the figure, the use by all of the components of the Utilities component is not shown, nor is the use of the Error Handler by the Name and Update Handlers shown.

Figure 6-7: An Extended Architecture



The revised functionality of the Name Handler and Storage Handler is briefly outlined next. It corresponds mainly to the existing implementation but some projected additions are also indicated.

6.11.1 Name Handler

Before calling other components, relation names are checked for existence and then translated into internal identifiers. If they exist, then the required function is called, otherwise the Error Handler is called. Other parameters are checked at each function. Some functions use or return an object identifier (tuple id or relation id). They are the identifiers of the structure. In the case of tuples, a *tuple id* is a reference to a *tuple header* structure associated with the tuple. If the parameter does not include *id* then a reference to the structure itself is passed. In the case of tuples this is a reference to a vector of strings.

- **open database (database name)**: if a new database is opened, the necessary ICS structures are created (e.g. *relations* and *attributes* relations). It also performs some table initializations to speed up the mapping from names to internal identifiers. At this point, some security mechanisms may be invoked to check the access rights of the user to the database. The parameters of the function may be then extended to include a user identification and password.

- **close database (database name)**: called at the end of a session with a user interface and only if the user decides to *commit* any changes made to the database. It cleans up some temporal tables that may not persist. It is also feasible to collect final session statistics and restructure relations according to these statistics or upon explicit user request.

- **add tuple (relation name, tuple), delete tuple (relation name, tuple), modify tuple (relation name, old tuple, new tuple)**: calls the Update Handler to check protection, possible duplication of tuples and integrity constraints. If the Update Handler judges the operation as valid then the Storage Handler is called to process the operation. These, and other functions listed next, acting presently upon one tuple only, could be easily extended to work on all or a specified number of tuples.

- **find tuple (relation name, target tuple)**: the Storage Handler is called to find a tuple matching all the attribute values of the target tuple. This function may additionally accept a parameter indicating which attribute(s) ought to be consider(ed) in the matching test, and at which tuple to start the test.

- **get tuple (relation name, tuple id, which)**: **which** is any of first, last, next, or previous. The Storage Handler is called to get the required tuple.

- **store structure (structure name, structure reference), get structure (structure name), drop structure (structure name)**: a user interface may require from the Name Handler to make persistent any structure that may be required in another session. This is used in query languages to store views, snapshots, to leave a message that is recalled automatically in the next session, etc.

- **compile (query name, query string)**: the common Syntax Analyser is called to compile the query and, if correct, translate it into a tree which is returned to the calling interface.

- **evaluate (query name, query tree)**: the object names in the query tree are replaced by internal identifiers before the query is passed to the Query Evaluator.

- **cardinality (relation name), degree (relation name), keys (relation name)**: the Storage Handler maintains the cardinality and degree of each relation in a relation header. The keys of a relation are those attibutes that have the constraint *unique*.

- **create index (relation name, attribute name), delete index (relation name, attribute name)**: indexes are currently limited to one attribute and are implemented using the tables facilities (B-trees [Comer 79]) of PS-algol.

6.11.2 Storage Handler

This component performs the following operations:

- **add relation**: an empty relation structure is created, returning the new relation identifier.

- **delete relation(relation id)**: the relation structure and associated indexes are deleted.

- **add attribute (relation id, attribute type, attribute default**: each attribute has associated in its Storage structure a default value and a type that is required by the Query Evaluator.

- **delete attribute (relation id, attribute id)**: deletes the specified attribute.

- **add tuple (relation id, tuple), delete tuple (relation id, tuple id), replace tuple (relation id, old tuple id, new tuple)**: "replace tuple" corresponds to the **modify** tuple operation of the Name Handler.

- **find tuple (relation id, tuple), scan relation (relation id, tuple)**: the former searches the relation for a tuple matching exactly the target tuple, the latter checks only the non-null attribute values and is used by the Name and Update Handlers only.

- **sort relation (relation id, sort spec)**: this function is used by the Name Handler and Query Evaluator. The sort specification provides the identifiers of the participating attributes, the sort order and whether an ascending or descending order is required.

- **build index (relation id, attribute id), delete index (relation id, attribute id)**: this functions create a new index or destroy an existing one.

- **cardinality (relation id), degree (relation id)**: return the cardinality or degree of the specified relation.

- **max width (relation id)**: this function is used by the report generation facilities of each user interface. It returns a vector of integers where each element indicates the length of the largest attribute value, considering all tuples, for the corresponding attribute (attribute values are stored as strings in this implementation).

6.12 Logical Storage Handler Structures

Chapter 8 describes the research done to find suitable structures for relations and tuples. These structures are created, destroyed and maintained by the Logical Storage Handler. These structures, as well as the algorithms using them, can be provided on an incremental-complexity basis. The next section describes a set of structures used in an example relation.

A relation consists of a **relation header** structure, a vector of pointers to **attribute** description structures and a double linked list of **tuples** with the **attribute values** held in a vector of strings.

Some PS-algol structures used next are defined as follows:

```
structure pv (*pntr pv.p)
```

This structure embodies a reference to a pointer vector structure. In general, *T denotes a vector of type T, hence *pntr is a vector of pointers and *string is a vector of strings.

```
structure se (string se.s; pntr se.p)
```

This structure is a string-list element, that is, it contains a reference to a string (se.s) and a reference to another structure (se.p).

The Relation Header structure (rh) stands as the relation's identifier. i.e operations refer to a relation by means of its rh structure.

```
structure rh
( pntr     rh.ftup,      ! first tuple
           rh.ltup;      ! last tuple
  int      rh.degree,    ! degree
           rh.cardin;    ! cardinality
  bool     rh.temp;      ! true if relation is transient
  *pntr    rh.ad,        ! attribute description vector
           rh.sort,      ! pntr to se structures,
                         ! se.s="a" (ascending) or
                         !      "d" (descending)
                         ! se.p=pntr to attrib's ad str.
         rh.index )      ! pntr to index vectors
                         ! (tables & attributes)
```

Relations can be either **transient** (rh.temp=true) or **base** (rh.temp=false). The difference is in the outcome of the sort function which creates a new relation if the input relation is of type base. The sort is performed in the same relation if it is transient.

The Attribute Description Vector, contained in the relation's header (rh.ad), has pointers to attribute description structures (ad).

An attribute is identified by its position in the *rh.ad* vector.

```
structure ad
  ( string ad.default,    ! attribute default
           ad.type)       ! attribute type

The attribute type must be one of

  "i" : integer
  "s" : string
  "r" : real (not implemented yet)
  "b" : boolean ("T" for true, "F" for false)
  "d" : date (dd:mmm:yyyy)
  "t" : time (hh:mm:ss)
```

The attribute's default value **ad.default** must be of type **ad.type**. All attribute values are stored as strings.

A tuple is identified by its tuple header structure (tup), which points to the next and previous tuples (if they exist) and to the vector of strings containing the attribute values. Chapter 8 describes alternative tuple structures.

```
structure tup
  ( pntr     tup.next,          ! next        tuple
             tup.prev;          ! previous    tuple
    *string  tup.att )          ! attributes  vector
```

6.12.1 Example Relation Structures

The layout shown in figure 6-8 is a (somewhat simplified) example of a relation structure with the following characteristics:

- The relation is transient
- It has 4 tuples
- It has 3 attributes (2 of type "s", 1 of type "i" (with default values "a","x" and 0 respectively)
- It has 1 index table on the first and second attribute
- It is sorted on the 1st att. in ascending order and on the 3rd att. in descending order.

The notation in figure 6-8 has the following meaning:

ftup: pointer to the first tuple (tup structure), which in turn points to the vector of strings containing the first tuple: {"a","x",7}.

ltup: pointer to the last tuple (tup structure), which in turn points to the vector of strings containing the last tuple: {"m","s",9}.

degr: relations's degree: 4 attributes.

card: relation's cardinality: 4 tuples.

temp: boolean value indicating whether the relation is transient or not: true.

ad: reference to vector of pointers (*pntr rh.ad) to attribute description structure (structure ad).

sort: reference to vector of pointers (*pntr rh.sort) to sort description structures (structures se.s). The first element of this structure is a string indicating whether it is ascending ("a") or descending ("d") order, the second element is a pointer to the corresponding attribute description structure (structure ad). The lexical ordering is deduced from the position of an attribute's ad structure in the sort description vector (rh.sort).

index: reference to vector of pointers (*pntr rh.index) to each index pointer vector (idp; only one is shown). The index pointer vector points to the description structure of each of the attributes participating in the index. At present, only one attribute participates in an index. The first element of the index pointer vector points to the index table (idx tab) whose key is the concatenation of the attribute values participating in the index. The table maps the key to a vector of pointers (structure pv) with references to the tuples holding the attribute's values indicated in the key.

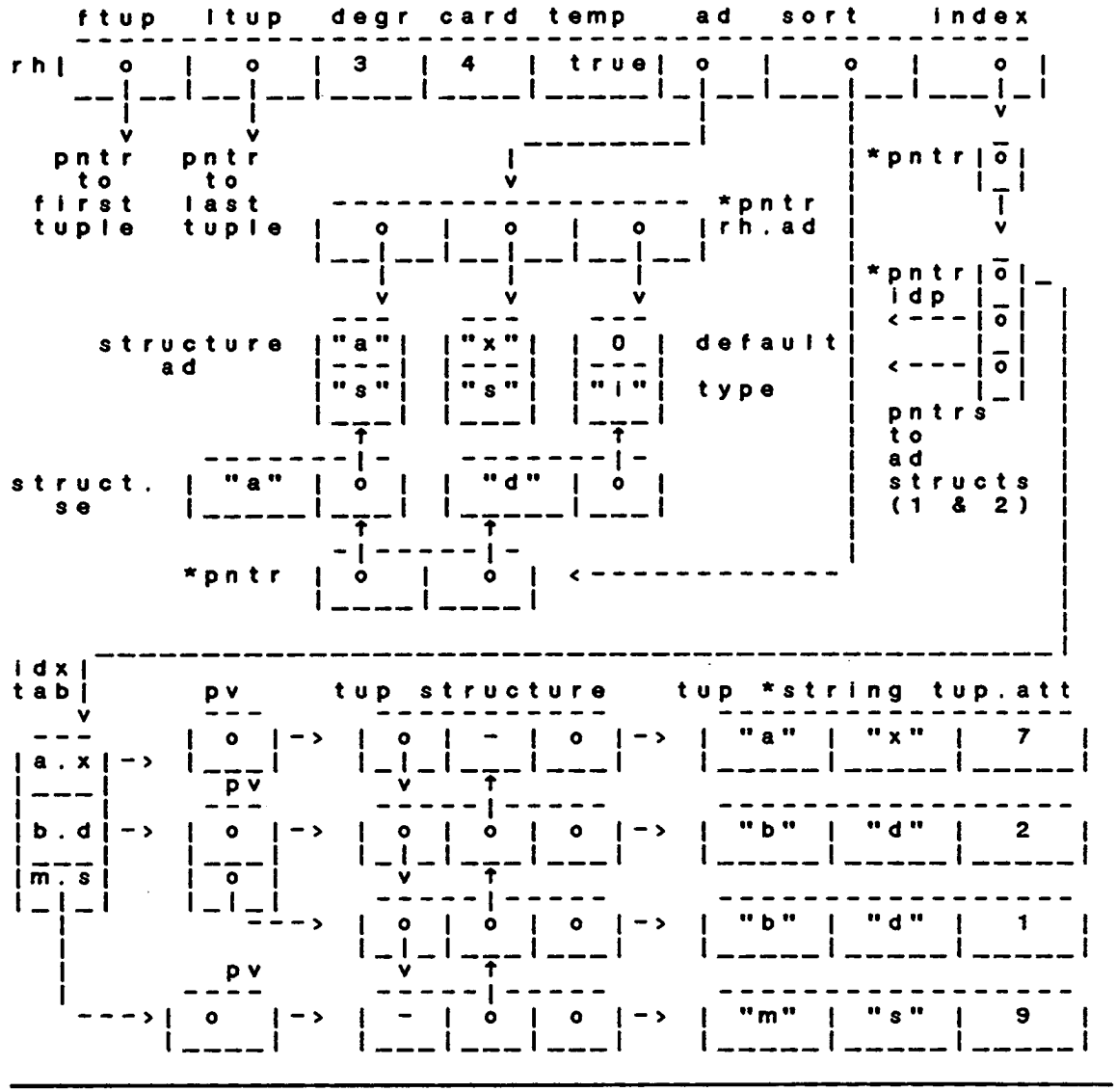
6.12.2 Logical Storage Handler Example

The following example program creates a relation with three integer attributes and five tuples. Since the objective is to illustrate a number of function calls to the LSH the example has been simplified, with function calls directly to the LSH instead of going through the Name Handler.

The LSH is embedded in a module which is called as an external procedure, indicating the function required (first (string) parameter) and the arguments required by the functions (next three (pntr) parameters). The LSH returns a reference to the result structure.

```
external lsh (string, pntr, pntr, pntr -> pntr)

structure tup (pntr tup.next, tup.prev; *string tup.att)
structure ba   (bool   ba.b)
structure sa   (string sa.s)
structure ia   (int    ia.i)
structure si   (string si.s; int si.i)
structure sv   (*string sv.s)
structure pv   (*pntr   pv.p)
structure iv   (*int    iv.i)

! Create a relation

  let rel := lsh ("addr", ba(false), nil, nil)

! Add 3 integer attributes to the relation, all with default = 0

  let att := nil
  for i = 1 to 3 do att := lsh ("adda", rel, sa("0"), sa("i"))

! Add 5 tuples with attribute values in ascending order

  let tuple.id := nil   ! tuple identifier
  for i = 1 to 5 do
  begin
    let tuple := vector 1::3 of ""
    for j = 1 to 3 do tuple (j) := iformat (i*j)
    tuple.id := lsh ("addt", rel, sv(tuple), nil)
  end

! sort relation on first attribute in descending order
```

Figure 6-8: Example Relation Structures




```
  rel := lsh ("sort", rel, pv(@1 of pntr [si("d", 1)]), nil)

! get maximum tuple width

  let max.width = lsh ("maxw", rel, nil, nil) (iv.i)

! print relation, adjusting attribute widths

  tuple.id := lsh ("get", rel, nil, ia(1))
  write "'n'n"
  for i = 1 to 5 do
  begin
    for j = 1 to 3 do
      write "| ", tuple.id(tup.att, j):max.width(j)," "
    write "|'n"
    tuple.id := lsh ("get", rel, tuple.id, ia(2))
  end
```

```
!-------------------------------------
```

The result of the program is the following:

```
|  5 | 10 | 15 |
|  4 |  8 | 12 |
|  3 |  6 |  9 |
|  2 |  4 |  6 |
|  1 |  2 |  3 |
```

6.13 Other Database Architectures

This architecture has some similarities with that of the database Handler of the programming language Plain [Kersten 81]. Plain is a Pascal-like programming language, designed to support systematic construction of systems that involve conversational access to databases by users. Unlike PS-algol, in which persistence is provided uniformly for all data types, Plain's only persistent type is *relation*. The language processor is connected through four layers to the underlying relational database system. The first layer is a communication interface which is similar to the Name Handler in that it controls the passing of messages between the Plain processor and the database Handler. The next layer is a logical database interface where database operations are interpreted, updates are checked and queries optimized. It corresponds to the common Query Evaluator and Update Handler. The next layer is the storage structure interface corresponding to the Logical Storage Handler. Plain's relation structures are B-trees only. The last layer is a page Handler.

Rowe and Stonebraker discuss a number of possible architectures for future databases [Rowe 81]. The assumptions made for future database systems are that they will have to cope efficiently with increased complexity and volumes of data, will be geographically distributed and shared, will probably use dedicated hardware in the form of back-end database machines and may be subject to standardizations because of marketing pressures rather than technological reasons. The architectures they discuss will have to support high level, non-procedural interfaces. The four architectures discussed are:

1. This architecture is shown in figure 6-9 and consists of a high level interface on top of an intermediate level (navigational) interface, such as Codasyl. Programs can be written for either interface, and a language processor invoked to translate the high level specification to a low level program. The two languages described in this chapter may fit in this description. RAQUEL corresponds to the end user interface and IQL to the intermediate level that can be made available to programmers. The Astrid system [Bell 80] provides a relational interface on top of a Codasyl system. In Astrid, queries are posed by the user in a relational algebra and the system automatically translates them into Fortran programs that run against the Codasyl database.

2. A high level interface on top of a low level access method interface, that retrieves one record at a time from the database. This may correspond to FQL with its own lazy evaluator connected to the Name Handler and Storage Handler, which is a low-level access method interface.

3. A high level interface on top of an interface which is not visible to an application programmer. This case corresponds to the Name Handler and RAQUEL Syntax analyser. The IQL language is not visible to user interfaces and the Name Handler may not be by-passed.

4. This architecture is shown in figure 6-10 and consists of a high level and an intermediate level interface on top of a low level interface. Two language processors are needed and the low level interface must handle both end user languages.

Figure 6-9: Architecture Case 1

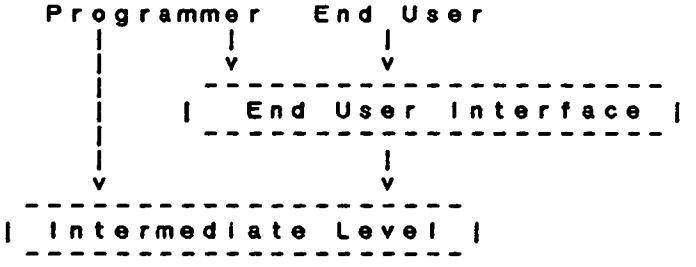
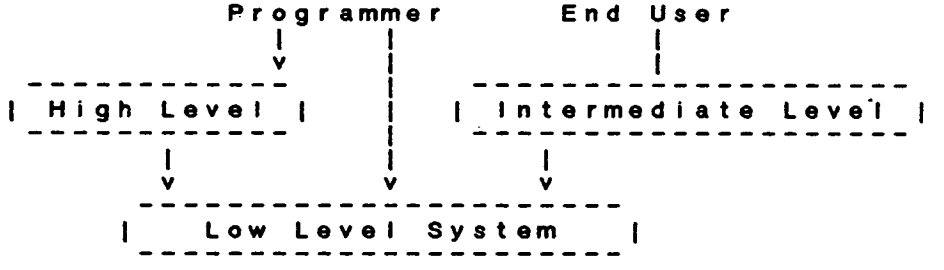


Figure 6-10: Architecture Case 4



Rowe and Stonebraker conclude that option 3 is the best if standards are not an issue and performance is tolerable. Its chances may improve with database machines. The problems in performance arise from the fact that this option has no visible access to a low level interface and it is assumed that the high level interface will produce less efficient code and manipulate less efficiently the retrieved data than a competent programmer would do. They calculate that the performance penalty will be around 15%. In our case, where it is expected that most of the requests from casual users will be simple, the code produced by the language translator should be nearly identical to that of a competent programmer. On the other hand, if performance is critical in an application where complex requests are issued, there is still the possibility of accessing directly the database through a dedicated PS-algol program. With respect to distribution, option three is the best since no low-level interface is visible and the language translator does not produce code to retrieve data in a record-at-a-time fashion. Clearly the RAQUEL and IQL may be both sent to other sites. The sending site need only collect the result data from each site and report to the user. This may be also the case with a database machine capable of processing IQL.

6.14 Conclusions

We presented in this chapter a series of components that can be gradually designed, implemented and tested. An important objective is to allow the users of the system to influence the construction of the system throughout its development. This is achieved by providing increasingly complex prototypes that evolve step by step into the final system. By defining suitable levels of functionality for each component, their evolution and that of the user interfaces can proceed in parallel. This was defined as the *minimal functionality* principle which is applied at each implementation stage and serves both to organize the design and implementation activities as well as a yard stick for each stage.

During our experiments in constructing database components we found that the rapid provision of new user interfaces can be achieved by constructing the internal components so that they allow the user interfaces to concentrate on their main tasks of collecting and requesting information to and from the user. This was defined as the *minimum responsibility* principle for user interfaces.

One of the aspects that helps faster building of a database environment is the identification of those tasks that are considered essential, postponing the addition of those less important to a later stage. Some of the traditional database tasks such as concurrency, recovery and security have not been incorporated. The reasons for this are threefold. First, we oriented our research toward personal or small-group databases where these tasks are less likely to become critical. Second, if they do become critical they can be incorporated to the system at a later stage, following the same principles outlined in this chapter. Third, some of them (e.g. concurrency and recovery) may be provided by the underlying software used to build the components (e.g. PS-algol will handle concurrency).

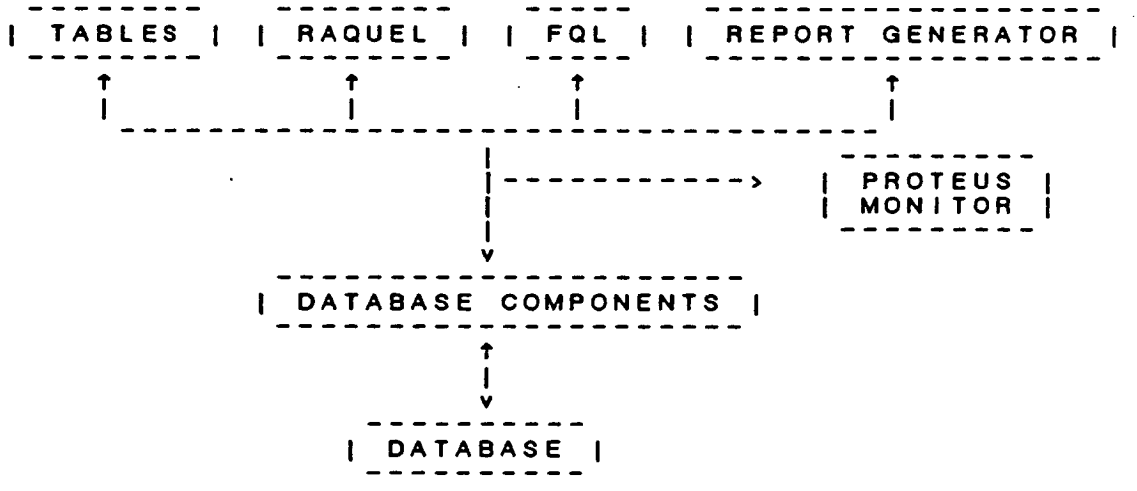
These components, resulting from our implementation experiments, allowed us to construct the database environment described in chapter 7 consisting of different user interfaces. An important finding of these experiments was that the database software can be effectively divided into separate but cooperating activities that can be gradually improved and extended.

Chapter 7
Implemented Database Environment

7.1 Introduction

This chapter describes the database environment implemented as a demonstration vehicle for the architectural propositions offered in previous chapters. All the user interfaces constituting the environment shown in figure 7-1 use the same set of database components, as described in chapter 6 and depicted in figure 6-7.

Figure 7-1: Implemented Database Environment



The user interfaces consist of two relational query and update languages, TABLES and RAQUEL, the functional query language FQL proposed in [Buneman 82], a Report Generator and an interface for the PROTEUS distributed databases project [Stocker 83]. All of them may access the same database with the novel feature that they use an entirely consistent name space. RAQUEL has been used by a number of undergraduate and postgraduate students during the 1982/1983 academic year. The other interfaces have been subject of demonstrations only so far. The whole set of interfaces described in this chapter is expected to be used by students in the coming academic year.

Although many people have used and commented on these interfaces and RAQUEL and the REPORT GENERATOR produced most of the results presented in chapter 9, we regard them, in their present state, as a demonstration of the reusability of the proposed components rather than complete user interfaces. More development is both sensible and necessary. For this reason and while outlining their present state, we propose some desirable improvements on them.

We have advocated a user oriented approach to database software engineering throughout this thesis. For that reason, we considered that by observing the way query languages are used we could improve their appeal to the users. Chapter 9 shows the results of observing the use of EDQUSE [Wilson 81] and of RAQUEL. The EDQUSE results influenced the design of the TABLES and RAQUEL user interfaces in many ways. We had the opportunity to assess this influence during a database exercise set to third year Computer Science students (using a more primitive version of RAQUEL than the one presented in this chapter). Given a choice of RDB (a functional/relational query language [Nikhil 82]), EFDM (a functional query language [Kulkarni 83]), ASTRID (a relational algebra query language [Bell 80]), EDQUSE and RAQUEL, out of 14 groups of three students each, and when TABLES was not available, one chose RDB, one chose EDQUSE and 12 chose RAQUEL to do a variety of database projects. In an earlier database exercise for a group of MSc students, the same languages were analysed by them, deciding finally to use RAQUEL for their project.

The user interfaces are described in general in this chapter, using the database shown in figures 4–1 and 4–2 for examples. Further details of the interfaces are given in a set of manuals [Hepp 83].

TABLES is expected to be used by casual users as well as by those requirying only limited processing capabilities. RAQUEL is a relational algebra that uses a more elaborate mechanism than TABLES to build queries and is fully relational. The implementation of FQL is better defined as a "browser" than a query language since it is not able to retrieve queries involving more than one relation. The Report Generator is an enhanced output facility from the one provided by default by all the query languages. It is also capable of peforming some basic statistics on relations as well as drawing histograms.

In the following presentation of the user interfaces, which is a summary of the manuals, the words *table*, *column* and *row* are sometimes substituted for *relation*, *attribute* and *tuple* respectively, since they have been used in the manuals.

7.2 TABLES

TABLES is a screen oriented query and update language. The design of the screen handler has drawn heavily from EDQUSE [McDuff 80] which in turn has borrowed many ideas from QBE [Zloof 75]. TABLES is expected to compete with EDQUSE in terms of being easy-to-use.

TABLES models applications as a set of named tables (relations). Each table consists of a set of rows (tuples) and a number of named columns (attributes). All values in a column have the same type which can be one of integer (i), string (s), boolean (b), time (t) or date (d).

Every new database created through TABLES is provided with three tables. These are the *tables*, *columns* and *help* tables. The first two are common to all relational query languages accessing this database and contain information on table names, column names, column types, etc. The latter is particular to TABLES and provides a summary and examples of its use. The *tables* and *columns* tables constitute the TABLES conceptual schema and may be queried and updated as any other database query, provided some associated constraints are satisfied. This is a major difference from EDQUSE which provides no visible schema and provides different mechanisms for adding new tables and columns.

With respect to help facilities (EDQUSE has none), TABLES provides the *help* table plus short summary messages while forming queries, traversing or updating a table.

The following is a layout of the *tables* table after a new database has been created i.e no user created table exist.

tabname	sortorder	protection
s	s	s
columns	tabname=a	dt,ac,dc
help	n	n
tables	tabname=a	dt,ac,dc

The first row is the table header and contains the *column names*. In the above table, these are *tabname*, *sortorder* and *protection*. The next row indicates the column types which in the above case is *string* ("s") for all columns. The subsequent rows contain column values. The fourth row indicates that there is a table whose name is *help* and that it is not sorted and has no protection.

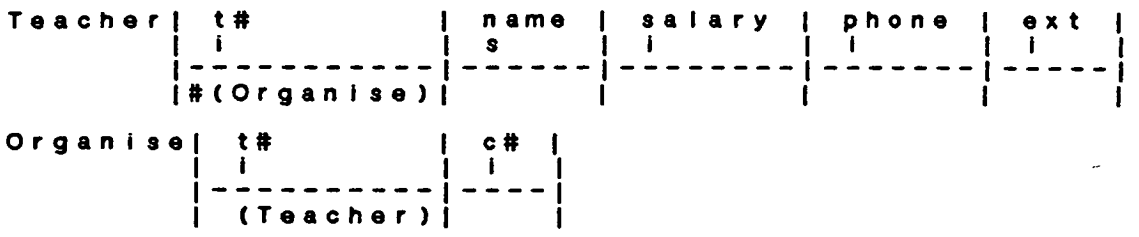
The *columns* table contains the table name, column name, type, default, constraint and position of all columns of all tables in the database. Unlike EDQUSE, TABLES allows the user to define integrity constraints associated with column values, protection to tables, columns and rows, default values and sort order. The meaning and use of each of these columns is explained in the presentation of RAQUEL.

In addition to the above structures, a user may define *views* which are named queries that persist between TABLES sessions. The view mechanism is explained later. This facility does not exist in EDQUSE and may significantly enhance the usability of the interface for users defining large and repetitive queries that may persist as "views".

7.2.1 TABLES Queries

Like QBE but unlike EDQUSE, TABLES queries have a name and can be used in building further queries. This is an important advantage over EDQUSE in which a complicated query cannot be broken into smaller queries. Furthermore, we have observed that understanding and managing queries becomes difficult if the query involves more tables than the screen can hold at one time (5 tables at most). With TABLES, a query can be systematically built out of simpler queries involving a screen full of tables each time.

Figure 7-3: TABLES Join Query



7.2.2 TABLES Commands

A TABLES session starts in *command mode* from which a number of commands can be used, including some that will enter another mode, from which the user may return by providing a "null" command (press RETURN key in response to the command prompt).

To display a table on the screen, the user types the name of the table which is then displayed without any interposed evaluation. With EDQUSE, the user must define a query over the table (possibly projecting all columns), which is then analysed, evaluated and displayed. The simpler approach used in TABLES may save a considerable amount of time since during a session, existing tables or query results may be re-displayed many times. This limitation of EDQUSE may account for the large proportion of "project only" queries recorded (see chapter 9).

Once a table has been displayed, the user enters *traverse mode*. A cursor is positioned on the first row, first column of the table. The following commands describe those available in TABLES when in the traverse mode.

- bottom : move cursor to the last row of the table displaying new rows if necessary.
- cfind : (find value in column) search the table for a value in the current column that matches the given value. Position the cursor in that row. If no value is found, leave the cursor at the bottom row.
- down : move cursor to the next row, if any.
- find : as cfind but search in all columns.
- help : display names of commands that can be used in the traverse mode.
- left : move cursor to the column at the left, if any.
- right : move cursor to the column at the right, if any.

- top : move cursor to the first row in the table, re-displaying rows if necessary.

- up : move cursor one row up, if any.

The following list describes the commands available in TABLES when in command mode:

- exit : terminate TABLES session. The user decides whether to *commit* or not. If the answer is positive, any updates performed during the current session will be applied. Otherwise the database preserves the state it had before the start of the session.

- drop : a given "view" name is dropped from the views table.

- help : display the *help* table.

- include : a given query name is included as a view.

- load : bulk load of rows from an external file.

- queries : display the query names and the names of the tables included in each query.

- query: define a new query or edit an existing one. A query name must be given. If a query with that name exists, then some of the tables it includes (up to a screen full) are displayed as skeleton tables on the screen. The user may then use the following commands (listed in alphabetical order), to complete the specification of a query:

    - constant : define a constant element in the current column. The result query must have in this column, values that match the constant element. (This implements the "select" operation). An example was given in figure 2-2where the word "SCIENCE" was used as a constant.

    - down : move the cursor to the next row down. Create a new "empty" row if necessary.

    - forget : the user is asked for a table name that will be excluded from this query. A query may not have any tables isolated i.e. not joined to the other tables. To avoid that syntactic error, a table which has been obtained by get must be excluded by forget if it is not needed.

    - get : the user is asked for a table name whose skeleton will be displayed on the screen (if it is not already there) and the cursor positioned on its first column.

    - help : display message with available commands for queries.

- o join : used to join tables, as described in the example in section 7.2.1.
- o kill : delete the element in front of the cursor, if any.
- o left : move the cursor to the column at the left, if any.
- o next : move the cursor in front of the next element in the current column, if any.
- o print : insert a "p." element in the current column to indicate that this column must be included in the result table.
- o right : move the cursor to the column at the right, if any.
- o tables : list the names of the tables participating in this query.
- o up : move the cursor one row up. Stop before entering the row with column types.
- o =, >, <, <=, >=, <> : similar to constant (which defines equality), these commands define the relationship that must exist between the given element and the column values in this table, in the result table.
- o A "blank" response to the command prompt will terminate the definition or edition of this query and return to command level.

- output : produce a formatted output file from the given table or query name.
- update : display a given table name and enter update mode. This mode allows the use of all the commands defined for the traverse mode plus the insertion and deletion of rows and the modification of column values. Any update is applied immediately i.e. what the user sees on the screen is what is actually in the database. The user is not required to run a query on a table before updating it (as in EDQUSE) but the named table is directly displayed. The most important difference from EDQUSE is that in TABLES the update mechanism applies to any table in the database, including those constituting the schema (the tables and columns tables). A new table is added to the database by adding a row to the *tables* table. A similar procedure is used to add new columns.
- views : display current views and the names of the tables involved in each view.

7.2.3 Mapping TABLES Queries into Relational Algebra

After a query is defined or edited, TABLES parses the query tables generating a relational algebra expression that is passed to the common syntax analyser which in turn generates the Internal Query Language (IQL). The algebraic expression is actually a RAQUEL expression as

defined in the next section. To achieve the translation from table tokens to a relational algebra expression, the data structures used in TABLES for representing a query and the mapping process are sketched next.

Each query consists of a PS-algol structure including, among other fields, a string field to contain the corresponding relational expression and a vector of pointers field where each table in the query is represented by one vector element pointing to a table structure. The table structure contains the name of the table, the name and type of each column in the table and a 3-dimensional (3-d) structure pointing to table element structures. The 3-d structure can therefore hold references to any table row (one dimension), any column in that row (second dimension) and any element of that column, same row (third dimension). Recall that a table in a query may contain many columns and each column may contain more than one element e.g. a "print" element, constants and join table names, as shown in figure 7-4.

Figure 7-4: TABLES Table with Multiple Columns and Elements

Student	s#	name	faculty
	i	s	s
	5	"Leonor" p. (Attend)	"SCIENCE"
	6	"Magdalena"	"ARTS" p.

Each Table element structure has three fields. One indicates the operation associated with the element (i.e. one of project, select or join), the second field contains the value that is actually displayed (e.g. the "p." command, a constant or a join table name). The third field is used if the first element field has a join operation, to indicate the column used for joining the other table.

The query's relational expression is constructed by examining each table's 3-d structure, starting from the first table in the query vector field. Each table will produce a (possibly null) "select" operation associated with it. The figure 7-4 would produce the following selection on the table "Student": Student selected with ((name="Leonor" or name="Magdalena) and (faculty="SCIENCE" or faculty="ARTS")). At each element with a join operation, the 3-d structure of the other table involved in the join operation is examined to check and complete the operation. If a "project" operation is found ("p."), it is added to a global project list which is appended to the final relational expression.

7.2.4 TABLES Potential Role and Improvements

Although TABLES offers many advantages over EDQUSE, there are still many improvements that could be made to it, especially for casual users.

One of the problems with non-procedural languages such as this is that it is sometimes difficult to mentally construct the result table while formulating the query. It would be therefore desirable to display the structure of the result table while constructing the query. In the long run one might consider "debugging" aids for queries such as being able to identify a

column of this result table and have a trace of where its data comes from shown on the screen.

The screen oriented approach of TABLES may be greatly enhanced once better graphics capabilities become available. For example, a Bachman diagram may be made available with each database schema. The joining of tables could be also improved. For example, once a person gets a table, all other tables that may be joined with it could be also displayed and the joins filled automatically. The user would then have an immediate picture of all possible relationships which he can establish and kill those unwanted, using perhaps a "mouse" device. One can also envisage layout operations being associated with each table such as user defined screen-masks for displaying a table or updating it, as in LogiQuest [Logiquest 82].

Another area of potential expansion is the provision of a powerful screen editor for traversing and editing a table, including string and substring handling facilities for searching.

Finally, an authorization mechanism including passwords and personal access rights to selected portions of a database should be also provided. This is related to facilities for defining name spaces that have associated access rights. The notion of "view" that we have used refers to a different perception of one or more tables achieved by maintaining query definitions as persistent objects. The notion of view should be extended to a set of names (as used in file systems), defined and maintained by the users, which may be also considered as external schemas. This may be particularly important in large databases having a variety of tables on different areas (e.g. in the University database example, portions of the database may concern teachers only, others to secretaries, the accounting department, administration data, etc.). To achieve this notion of views, it will be necessary to design a "Views" component that maintains the mappings from the ICS to the external schemas. It may be first provided by defining SELECT operations over the ICS tables, defining the portion of the database that is required. These operations are incorporated by the Views component to any reference to the SELECTed tables to and from the user interfaces (e.g. in queries).

The principal aim of improving TABLES should be to increase the usability of database technology, especially for non-expert users. TABLES, and its underlying architecture, may serve as a basis for developing interfaces for applications where non-expert users require simple to use tools for organising and administering data.

The TABLES interface was designed once RAQUEL and all of the underlying components had been built. It was therefore possible to concentrate on the two main tasks of screen managing and query mapping. Both were accomplished in less than a week. This compares well with the time required to produce the complete EDQUSE system, which has similar screen capabilities but is less powerful than TABLES. EDQUSE was produced by two fourth year Computer Science students as their final honours project in consecutive years. Although EDQUSE involved the creation of the whole DBMS supporting it, and TABLES only the user interface, we would like to emphasize that the speed in producing TABLES was mainly due to the availability of an adequate underlying set of components as well as an appropriate programming language.

7.3 RAQUEL

RAQUEL is an interactive program with facilities for querying, updating and manipulating a database. It is a relational algebra and its design has been influenced mainly by Astrid [Bell 80]. TABLES users may regard RAQUEL as the natural next step in terms of processing power since both RAQUEL and TABLES can access the same database and both use a consistent name space. Compared to TABLES, RAQUEL is more suitable for expert users, although it has been successfully used by non-experts as indicated in the introduction. Its main advantage over TABLES relies in its greater processing capabilities. It is fully relational and has grouping functions. It has uniform access to both data and meta-data.

As in TABLES, a database is organised as a collection of named tables, three of which are provided with every new database. these are the *tables*, *columns* and *helpraquel* tables.

The *helpraquel* table contains information on the commands and operations that can be performed in RAQUEL.

Data Types and Type Operations: the data types supported by RAQUEL are: integer (i), string (s), boolean (b), time (t) and date (d).

The following relational operators may be used:

```
=  : equal to
<  : less  than
<= : less  than or equal to
>  : greater than
>= : greater than or equal to
<> : not equal to
```

The order for strings is lexicographic, for booleans is "F" < "T".

Integer operations:

```
+,-,*  addition, subtraction, multiplication,
/,rem  division, remainder.
-      sign invertion.
```

String operations:

in: Substring e.g. "bc" *in* "abcde" yields true

++: Concatenation e.g. "abc" ++ "de" yields "abcde"

Boolean operations:

and, or, not

Notation: a *name* is an arbitrary length sequence of letters , digits and the characters '#' and "_".

The syntax of the operations on tables is explained through examples and also using the following more formal notation:

- all symbols are introduced as expressions where the left-hand side introduces a name for a symbol and the right-hand side defines the symbol.

- non terminal symbols (defined in terms of other symbols) are specified as identifiers and terminals as string constants (between apostrophes).
- names to be defined by the user are stated between angle brackets.
- alternatives are separated by a vertical bar.
- a right-hand side expressions may contain braces to delimit expression which can occur zero or more times.

Example. positive numbers can be represented as:

```
number ← digit | number digit
digit  ←   '0' | '1' | '2' | '3' | '4' |
           '5' | '6' | '7' | '8' | '9'
```

```
alternatively a number may be defined as:

number ← digit { digit }
```

7.3.1 RAQUEL Queries

A query is defined through the command "query". Queries are terminated by a semicolon and may be spread over several lines.

A query produces a temporary table i.e. it is not included in the *tables* table and cannot be updated. Note that since a query defines a table, previous queries may be used when defining further queries. This is very useful in building complex queries in terms of simpler ones.

The operations are of type *unary*, operating over one table or *binary* operating over two tables. One query definition may invoke many operations.

The flavour of RAQUEL will be presented in the following examples. "C:" is the RAQUEL command prompt and the text at its right is entered by the user. If the text starts with the character "!" then it is a comment and is not processed. A table (or a query result) is displayed by typing its name in response to the command prompt.

Unary Operators.

The Project operator forms a vertical subset of an existing table. Redundant duplicate rows in the result table are removed. An example is shown in figure 7-5.

```
The syntax is:

project  ← <table-name> proj-op col-list
proj-op  ← any abbrev. of word 'projected'.
           The word 'to' is optional
col-list ← <column-name> {',' <column-name> }
```

The Select operation forms a new table by taking a horizontal subset of an existing table, that is, all rows of an existing table that satisfy a given condition. An example is shown in

Figure 7-5: Raquel: Project Operation Example

```
C: ! project the tabname and colname columns
C: ! from the tables table producing a result
C: ! table named tabcol:
C: query tabcol := tables projected to tabname, sortorder;
C: tabcol

tabcol:
 ------------------------
| tabname    | colname   |
| s          | s         |
|------------|-----------|
| columns    | tabname=a |
| helpraquel | n         |
| tables     | tabname=a |
 ------------------------
```

Figure 7-6: Raquel: Select Operation Example

```
C: ! Produce a table named onlytab that contains
C: ! information about all columns of the "tables" table.
C: query onlytab := columns selected on tabname = "tables";
C: onlytab

onlytab:
 ------------------------------------------------------------
| tabname | colname    | type | default | constraint |
| s       | s          | s    | s       | s          |
|---------|------------|------|---------|------------|
| tables  | tabname    | s    |         | unique     |
| tables  | sortorder  | s    |         | n          |
| tables  | prtection  | s    |         | n          |
 ------------------------------------------------------------
```

figure 7-6. Note that in the formulation of the query, "tables" is a column value (not a column name as tabname) of type string and therefore must be presented between quotes. Parenthesis may be used to clarify evaluation order.

```
The syntax is:
select      ← <table-name> select-op expression
select-op   ← any abbreviation of the words
              'selected', 'where' or 'with'.
              The word 'on' is optional
expression  ← or-exp 'or' and-exp | and-exp
and-exp     ← and-exp 'and' ineq-exp | ineq-exp
ineq-exp    ← add-exp ineq-op add-exp | add-exp
ineq-op     ← '<' | '<=' | '>' | '>=' | '=' |
              '<>'| 'in'
add-exp     ← add-exp add-op mult-exp | mult-exp
add-op      ← '+' | '-' | '++'
mult-exp    ← mult-exp mult-op factor | factor
mult-op     ← '*' | '/' | 'rem'
factor      ← primary | sign primary
sign        ← '+' | '-' | 'not'
primary     ← atomic | '(' exp ')'
atomic      ← const | ident
const       ← integer-const | bool-const |
              string-const
integer-const ← digit {digit}
digit         ← '0' | '1' | ... | '9'
```

```
bool-const      ← "T" | "F"
string-const    ← '"' string '"'
string          ← char {char}
char            ← 'a' | .. | 'z' | 'A' | .. | 'Z' |
                  '#' | '_'
```

The **Modify** operator is used to modify column values in rows for which a given predicate is true. The expression and the column must have the same type. It is normally used in update operations to modify an existing table but it may also used in queries, creating a result table, as shown in the example of figure 7-7.

Figure 7-7: Raquel: Modify Operation Example

```
C: ! Create a table modifying the extension number to "77"
C: ! for those teachers having extension number = "0"
C: query new_ext := Teacher modified on
                    if ext = 0 then ext := 77;
C: ! show the Teacher and the new_ext tables
C: Teacher

Teacher:
 ------------------------------------------
| t#  | name        | salary | phone | ext |
| i   | s           | i      | i     | i   |
|-----|-------------|--------|-------|-----|
| 1   | Alicia      | 35000  | 24    |  0  |
| 2   | Margarita   | 24000  | 23    | 21  |
| 3   | Adriana     | 18000  | 22    |  0  |
 ...   .......      .....    ..      ..

C: new_ext

new_ext:
 ------------------------------------------
| t#  | name        | salary | phone | ext |
| i   | s           | i      | i     | i   |
|-----|-------------|--------|-------|-----|
| 1   | Alicia      | 35000  | 24    | 77  |
| 2   | Margarita   | 24000  | 23    | 21  |
| 3   | Adriana     | 18000  | 22    | 77  |
 ...   .......      .....    ..      ..
```

```
The syntax is:
modify     ← <table-name> modif-op if-exp
modif-op   ← any abbrev. of words 'modify'
             or 'modified'. Word 'on' is optional
if-exp     ← 'if' expression 'then' assign
assign     ← <column-name> ':=' expression

see select operation for definition of expression.
```

The **Order** operator is normally the last in a query definition. It produces a result table ordered by column values in either ascending or descending order. An example is shown in figure 7-8.

```
Syntax:
order     ← <table-name> order-op ord-spec
ord-spec← <column-name> '=' ord
            {,<column-name> '=' ord}
order-op← any abbrev. of word 'ordered'.
            The word 'on' is optional.
ord       ← 'a' | 'd'
```

Figure 7-8: Raquel: Order Operation Example

```
C: ! produce a table with the teacher's name in ascending order
C: query neat := Teacher ordered on name = a;
C: neat

neat:
 ------------------------------------------
| t#  | name        | salary | phone | ext |
| i   | s           | i      | i     | i   |
|-----|-------------|--------|-------|-----|
| 3   | Adriana     | 18000  | 22    |   0 |
| 1   | Alicia      | 35000  | 24    |   0 |
| 2   | Margarita   | 24000  | 23    |  21 |
 ...   .......       .....    ..      ..
```

The Extend operator is used to extend a table by appending a new column whose values are determined by an expression. The <col-list> indicates which columns from the argument table must be included in the result table (as in the project operator).

The expression can be formed using the same operators as with the select operator. The type of the expression determines the type of the new column. An example is shown in figure 7-9.

```
Syntax:
extend   <- <table-name> ext-op  ext-exp
ext-op   <- any abbrev. of word 'extended'
              the word 'to' is optional
ext-exp  <- col-list ':' <new-col-name> ':=' exp
col-list <- <column-name> {',' <column-name> }

See select operation for definition of 'exp'
```

Figure 7-9: Raquel: Extend Operation Example

```
C: ! produce a table similar to "Teacher" with
C: ! name and salary of each teacher and a new
C: ! column showing its salary increased by 1 %
C: query new_salary := Teacher extended to
     name,salary : new := salary + (salary / 100);
C: new_salary


new_salary:
 ------------------------------
| name        | salary | new   |
| s           | i      | i     |
|-------------|--------|-------|
| Alicia      | 35000  | 35350 |
| Margarita   | 24000  | 24240 |
| Adriana     | 18000  | 18180 |
  .......       .....    .....
```

The example in figure 7-9 shows that the extend operator is a combination of the project and modify operators, but instead of modifying the values in an existing column, a new column is produced. As in all other operations, a query does not modify the values of the tables involved in the query.

The Group operator [Gray 82] is used to partition the argument table into row groups such that within each group all rows have the same value in the column(s) indicated in the col-list and then to compute an expression over each group of rows. The expression should contain aggregate functions from the following list:

group function	expression type
count	-
sum (expression)	integer
max (expression)	integer
avg (expression)	integer
min (expression)	integer
all (expression)	boolean
any (expression)	boolean

all and any are useful in forming quantified queries. all computes the boolean conjunction for each group (the result is true if all members in the group are true) and any computes their disjunction (which is true if any is true).

```
Syntax:
group     ← <table-name> group-op group-exp
group-op  ← any abbrev. of word 'group'.
            The word 'on' is optional.
group-exp← col-list ':' <new-col-name> ':=' expr
col-list  ← <column-name> {,<column-name>}

expr : expression as in select operation.
```

The col-list indicates the columns to be considered to form the row groups. <new-col-name> is the name of the new column which will contain the value of the group expression. An example is shown in figure 7-10. In order to make it easier to update the meta-data, a notation for a table constant was introduced. It can be seen as a one-row table, with no name, and with the column types defined by the column values. Examples are shown in the update operations of section 7.3.3.

```
Syntax:
  const-exp ← '{' column-value-list '}'
```

Binary Operators.

The Join operation. If two tables have a common column type then they may be joined over those columns. The result of the join operation is a new table in which each row is formed by joining together two rows, one from each of the argument tables, such that the two rows concerned have the same values in the common column.

The join column from the right-hand side argument table is not included in the result table. Duplication of column names in this table are avoided by prefixing with "#" the names of the columns from the right-hand side table that also exist in the left-hand side table. An alternative procedure, not yet implemented, is to force a renaming before the join if the tables have common column names other than those participating in the join. An example is shown in figure 7-11.

```
Syntax:

join ← <table-name > j-op j-exp <table-name>
```

Figure 7-10: Raquel: Group Operation Example

```
C: ! Count the number of students in each faculty
C: query students_in_faculty := Student grouped on
       faculty : total := count;
C: ! display "Student" and "students_in_faculty" tables
C: Student

Student:
 ---------------------------
| s#  | name         | faculty |
| i   | s            | s       |
|-----|--------------|---------|
| 5   | Leonor       | SCIENCE |
| 6   | Magdalena    | ARTS    |
| 7   | Veronica     | ARTS    |
| 8   | Andrea       | SCIENCE |
| 9   | Alejandra    | SCIENCE |
 ---------------------------

C: students_in_faculty

students_in_faculty:
 ------------------
| faculty | total  |
| s       | i      |
|---------|--------|
| ARTS    | 2      |
| SCIENCE | 3      |
 ------------------
```

Figure 7-11: Raquel: Join Operation Example

```
C: ! join the tables column and tables by their common
C: ! column tabname.
C: query tab_col := tables joined through tabname = tabname columns;
C: tab_col
(only part of the result table (tab_col) is shown)

 ------------------------------------------------------------
| tabname    | sortorder  | protection  | colname ...
| s          | s          | s           | s
|------------|------------|-------------|------------
| columns    | tabname=a  | dr,aa,da    | position
| columns    | tabname=a  | dr,aa,da    | constraint
  ......
```

```
j-op ← any abbreviation of the word 'joined'.
j-exp← <column-1> '=' <column-2>
```

The Outer Join operation is similar to *join* with the difference that for those rows in the left-hand side argument table which do not have a matching row in the right-hand side table, a *dummy* right-hand side row is created with default values. An example is shown in figure 7-12. The above example illustrates several aspects of the outerjoin operation. First, the result table has at least the cardinality (in this example the same) of the left-hand side table x. Non-matching rows from this table are included in the result table z by appending default values ('0' in our example) in the columns of the other table (e.g. {1,5,0} and {3,7,0}).

Figure 7-12: Raquel: Outerjoin Operation Example

```
C: ! Consider the table z resulting from
C: ! outerjoining the tables x and y.
C: query z := x oj a=a y
C: x

x :   ---------
      | a | b |
      | i | i |
      |---|---|
      | 1 | 5 |
      | 2 | 6 |
      | 3 | 7 |
      ---------
y :   ---------
      | a | b |
      | i | i |
      |---|---|
      | 2 | 9 |
      | 5 | 8 |
      ---------
z :   -------------
      | a | b | #b |
      |---|---|----|
      | 1 | 5 |  0 |
      | 2 | 6 |  9 |
      | 3 | 7 |  0 |
      -------------
```

Second, the addition of non-matching rows applies to the left-hand side table only: it is a unidirectional outer join (note that row {5,8} of table y does not appear in the result table z). Finally, tables x and y have identical column names. To avoid duplicate names in the result table z, this table maintains the column names of table x, eliminating the join column of table y and renaming the other column b to #b.

Syntax:

```
o-join← <table-name> o-j-op j-exp <table-name>
o-j-op← 'oj' | any abbrev. of word 'outerjoined'

see join operation for the definition of j-exp
```

The **Set Union, Set Difference and Set Intersection** operations are the normal set operations. The tables must have the same number of columns (same degree) and the type of a column in one table must be identical to the type of the corresponding column in the other table. That is, the i-th column in both tables must be of the same type.

The column names of the result table are inherited from the column names of the left-hand side table.

```
Syntax:
set-exp← <table-name> set-op <table-name>
set-op ← '+' | any abbrev. of word 'union'
         '-' | any abbrev. of word 'difference'
         '.' | any abbrev. of word 'intersection'
```

An example, using the same x and y tables of figure 7-12 is shown in figure 7-13.

Figure 7-13: Raquel: Set Operations Example

```
C: query setunion := x + y
C: query setdiffe := x - y
C: query setinter := x . y
```

7.3.2 RAQUEL Commands

As in TABLES, RAQUEL provides a view mechanism. Views are used to provide a different appreciation of the database than the one provided by default by RAQUEL. It can also be used as an alternative naming system and as a means for maintaining the definition of frequently used queries.

Consider a query that is often used throughout sessions. To avoid redefining it each session, the query may be stored as a *view*, which will make it readily available in further sessions.

The following list describes the RAQUEL commands:

- *include* <qry> : include a given query as a permanent view.
- *drop* <view> : delete an existing view.
- *views* : display current views.
- *query* : used for defining a query
- *update* : used for defining an update
- *load* <file> : add rows from external file <file> to an existing table.
- *dump* <file> : dump content of a table into external file <file>
- *output* <file> : formatted output of a table to file <file>.
- <table-name> : display a table on the terminal.
- *queries* : display queries and their definition.
- *exit* : terminate session. The user is asked whether to commit or not. If the answer is positive, any changes made to the database during the session will be applied.
- *keepr* <table name> : used to save a result table as a snapshot. The table is made persistent, allowing the Report Generator to use it.

- *dropr* <table name> : the persistence of this table is not required any more. This command is related to *keepr*.

- *!* : a command line starting with a "!" character is not processed.

7.3.3 RAQUEL Updates

Adding Rows

```
Syntax:
update  ← <table-name> ':=' union-op  table-exp
union-op← '+' | any abbreviation of word 'union'
```

table–exp is any expression using table operations.

Example: add tables 'x' and 'y' to the database.

```
C: update tables := union {"x","n","n"};
C: update tables := union {"y","n","n"};
```

In the above example the table constant notation {..} was used as a table expression. In the same form, new columns may be added to a table by updating the columns table. Example: add two integer columns named 'a' and 'b' , with default '25' and '–1', to the 'x' table:

```
C: update columns := union {"x","a","i","25",""};
C: update columns := union {"x","b","i","-1",""};
```

Deleting Rows is symmetric to adding rows. The operation used is set difference.

```
Syntax:
delete ← <table-name> ':=' diff-op table-exp
diff-op← '-' | any abbrev. of word "difference"
```

Modifying Rows The syntax is identical as for the query case. The difference is that the modification will affect a (permanent) table. Example: assume that a table 'x' has in column 'a' values that are < 0. Modify them to be equal to the sum of the values in columns 'b' and 'c'.

```
C: update x := modified on if a < 0 then a := b+c;
```

Note that the modify operation is similar to the extend operation, the difference being that extend creates a new column and modify modifies an existing one.

7.3.4 Protection, Constraints and Defaults

PROTECTION is defined at the table level (in the tables table). By default, a table is unprotected (protection="n"). The protection may be defined when creating a new table i.e. adding a row to the tables table or by modifying a row in the tables table. A table can be protected against the following events:

```
dt : delete table
ac : add column
```

```
dc  :  delete  column
at  :  add  row
dr  :  delete  row
mr  :  modify  row
```

At the column level, a column can be defined as having one of three possible *CONSTRAINTS*. RAQUEL will check that during a loading or update operation, the constraints are not violated. If they are, the offending row will be neither updated nor loaded but the operation will continue.

- *"unique"*: the column value cannot be duplicated in the same column in any other row in the table.

- *isin* <table–name>.<column.name> : Referential integrity constraint. It specifies that the column values in this column must exist in some row, in the given column of the given table (see the *columns* table for examples).

- *<Predicate>* : any boolean expression that must yield true when adding a new value.

Example of predicate constraint when adding a column to the *staff* table:

```
C: ! Add an 'age' column to the "teacher" table,
C: ! where  0 <= age <= 150
C: update columns := u {"Teacher","age","i","25",
            "(age>=0) and (age<=150)"};
```

DEFAULT values for columns are defined when creating them. All columns must have a default value which will be applied, if necessary, when updating a table (e.g. adding a row without knowing all column values needed). In the previous example, the column age was given the default value '25'. All existing rows in the table are automatically extended to include this column value. If a default value is not supplied when defining a column, RAQUEL provides one for each type: '0' for integers, the null string "" for strings, 'false' for booleans and the current date and time for these types.

7.3.5 RAQUEL Role and Improvements

As TABLES, RAQUEL offers an open–ended area for improvements and extensions. A number of additions such as adding more operations to each of the data types could be easily introduced. It has been suggested by most of its users the addition of more extensible string and substring handling facilities e.g. string matching without reference to case, etc. It is also possible to include new relational operations such as discard, similar to project but excluding the named columns in the result table and rename to assign new names to column names. The operations extend and modify could be merged into a single operation.

New grouping functions may be included (e.g. to concatenate string values in the group) or extend the IQL to handle proper functions. Queries may be also defined as functions, substituting parameters at run time. This addition has been suggested by most of the RAQUEL users. The present notation for queries may be considered as functions without parameters.

The same view mechanism proposed for TABLES can be used for RAQUEL. Presently, view and query definitions are stored in PS-algol tables. A next version ought to keep them as database tables, subject to queries and updates.

RAQUEL does not presently define domains other than by predicates. A more proper handling of domains would include a domains table where each domain could be named and defined in different ways e.g. as presently through predicates, as the concatenation of other domains, etc. Domain definitions could be therefore shared among columns.

As in TABLES, many facilities could be added if more screen facilities, such as multiple-windows were available. For exmaple, the columns of the result table could be simultaneosly displayed while defining a query. The resulting query tree may be also displayed on request. As a debugging aid, the sources of the data in each of the columns of a result table, as well as the name transformations that have taken place could be displayed graphically. This could proceed, from the current query, back to the base tables e.g. if the query refers to tables resulting from other queries.

The protection mechanism was designed as a preliminary attempt to provide some protections. It could be improved in many ways e.g. by adding the notion of "person" to the database and associating protections with individuals or groups of persons. The integrity constraint mechanism provides also ample room for extensions e.g. defining unique over a set of columns instead of over one as presently. The present version is restricted to one constraint definition per column. A more proper handling of them could be achieved by defining a constraints table containing the table name, column name and constraint definition.

7.4 FQL

FQL was implemented as proposed in [Buneman 79] and [Buneman 82]. It has access to the same databases used by TABLES and RAQUEL but provides a functional view of it. It was constructed with most of the components also used in TABLES and RAQUEL. It has however, its own functional query evaluator. Although it is a "query only" interface, the no-updates problem may be circumvented by using one of the other languages.

The implemented FQL syntax described next corresponds to an earlier version, designed as an intermediate language and not intended for an end-user interface. The new syntax is described in [Buneman 83].

A distinguished feature of FQL is that functions can be defined independently of database objects.

Consider the following function definition:

```
let sum    := [1,2].+
let mult   := [id,10].*
let mult5  := 5.mult
let add1   := &[id,1].+
```

The "sum" function is equivalent to 1+2. The square brackets define a *tuple* function whose

components may be any function. For convenience, integers and strings are represented as functions too. The dot stands for composition of functions , that is, the composition of the "tuple" function and the "+" function. The type of the "sum" function is [int,int] -> int.

id is the identity function, that is, it returns its argument. The "mult" function is a partial functions that, given an argument, returns the argument multiplied by 10. e.g. mult5 will yield 50. The type of the "mult5" function is int -> int.

The function add1 is a sequence producing function that will yield an infinite sequence of numbers (the evaluating program may only generate the sequence on demand). The operator "&" is a generator that applies successive powers of the function to its argument.

e.g. 5.add1 will yield 6 7 8 9 10 11 12 .....

Functions over database objects are defined automatically in a functional schema. Every relation defines a constant function that will generate all the tuples of that relation.

e.g. !Student: -> *Student

the function "!Student" produces a sequence (*) of students, the function

!Student.name

produces a sequence of student names (it is of type *Student -> *String) and the function

!Student.|([faculty,"SCIENCE"].=).name

which is also of type *Student -> *String, produces a sequence of student names whose faculty is "SCIENCE". The operator "|" corresponds to the relational SELECT operation. It filters the members of the sequence generated by the function "!Student", according to the predicate [faculty,"SCIENCE"].=.

The mapping from relations and attributes to functions is outlined in chapter 4. We had previously implemented FQL in PASCAL. The PS-algol implementation required less than a third the volume of source code text.

FQL was designed and implemented after RAQUEL and TABLES and it demanded approximately one week of work. Three factors made this possible. One factor was the proper definition of the FQL language, its syntax and processing rules as well as the implementation indications given in [Buneman 82]. The second factor was the availability of a set of components that allowed us to concentrate on the user interface problems. The third factor was the facilities provided by the programming language PS-algol.

7.5 PROTEUS/Monitor

The PROTEUS/Monitor interface was the first prototype designed for this research. It was used to test the component's functionality as they were built and to test the IQL derived from the work by Atkinson and the author for the PROTEUS project (whose aim is to implement a distributed database system involving several U.K. Universities [Atkinson 82]). The program was demonstrated at a PROTEUS meeting, being one of the first software components of that

project in working condition. However, the IQL language for this research evolved independently from that of the PROTEUS project and so did the PROTEUS interface. Its main application was monitoring the evolution of the software components and the IQL, hence its name.

The Monitor is an interactive program which accepts queries written directly in the IQL language and processed by the Common Evaluator. It also accepts IDML commands to directly add, delete or modify tuples. It works on the same database used by all of the other user interfaces. Many early performance measurements of the PS-Algol language were produced by the Monitor but work on it was discontinued after RAQUEL became available.

Its most important contribution to this research was as its first testing tool for experimenting with software components that were incrementally build and are now used (reused) by all of the user interfaces described in this chapter.

7.6 Report Generator

The Report Generator is an interactive program that can access the same databases used by RAQUEL, TABLES or FQL. In addition to report generation facilities, this program also provides some basic statistic capabilities. The user specifies which database table is to be used for the report or the statistic analysis. Various commands, described below, may be used to shape the report by modifying a set of default parameters. The effect of each of the commands can be observed on the screen. This also applies to the statistic analysis.

As indicated in the introduction, during the academic year of 1983, third year students as well as MSc. students evaluated different query languages available in the department. One of their criticisms was the lack of suitable report generation facilities (except Astrid [Bell 80]). RAQUEL was part of that evaluation but it had only an elementary output facility, for both formatted and unformatted output.

We then decided to provide a more advanced output facility that could be used as another user interface. Query languages may ask the name handler to save the result of a query (a table), that is recalled by the Report Generator and manipulated by the user.

It also became apparent the need for some basic statistic facilities to process the data gathered from EDQUSE and RAQUEL user queries. This data was stored in a database and queried through RAQUEL using grouping functions to produce some basic statistics. Once more data was available, it became clear that the grouping functions were too limited. Histograms as well as output files that could be used with the graphics programs driving the departmental plotters were also needed.

Since the above report and statistics facilities ought to be used by all query languages, we decided to combine them in a single user interface that is described below. It is expected that its functionality will be gradually extended and finally derive into two user interfaces. One would be dedicated to report generation and the other to statistics and graphs. What is described next is only a small step towards that aim.

The Report Generator responds to the following commands:

- tables : display the database tables names i.e those in the tables table. Other tables may exists in the database that can be used for report generation e.g. those stored using the command keepr in RAQUEL.

- table <table name> : defines a table for report generation or statistic analysis. All further commands will apply to this table, until another table command is issued.

- title : defines the report title. The user can provide a set of textlines that will be inserted, alone, on the front page of the report.

- rrline <line number> : replaces report title line in current report title.

- ptitle : defines a page title. Every page of the report may begin with a set of textlines as defined by the user. All pages, except the front page will carry the same page title.

- rpline <line number> : replaces page title line in the current page title.

- header : defines page headers. Every page can be associated with a different page header that is printed before the page title. It consists of one line of text.

- rhline <line number> : replaces a page header line.

- chonly <column number> : print column value only when it changes, further identical values are replaced by spaces.

- chname <column number> : change specified column name in current table.

- chpage <column number> : start a new page each time the column value changes.

- display : display current table (report) on the screen. The table is formatted by default, i.e. all columns have the same length and values are justified to the left.

- plength <page length> : define page length in terms of number of rows in the table. Each page number will be indicated in the report.

- schar <new separator character> : change the current separator character. Column values in the report are separated by spaces and a separator character, which is by default equal to "|" but may be changed on request.

- uchar <new underline character> : changes the current underline character. The names of the columns (inserted on each new page) are separated from the table rows by an "underline" line. This underlining is also performed after the last row of the table. The character for underlining is "-" by default but may be changed on request.

- status : display status of report (table name, underline character, page length, etc.).

- help : display the name and a short explanation of each of the available commands.

- exit : finishes report generation session.

The statistic facilities work on a sample constituted by all the values of a column in the current table. These facilities are:

- basic <column number> : generates "basic" statistics using the specified column (which must be of type integer) of the current table. The "basic" statistics are: average, variance, range of sample, maximum, minimum and typical value. Number of occurences and percentage of sample are given where relevant (e.g. the maximum value in the sample, the number of times it occured and the percentage of occurence with respect to the size of the sample). These statistics are given for the whole sample and for the sample without the maximum and minimum values.

- histo <column number> : generates a histogram from the values of the specified column.

- graph : generates an output file for a graphic program. The graphic devices in the department can by used through dedicated programs that require data and parameters (e.g. graph title, units, etc.) in a particular format. This command produces a file from the current table, with such a format.

Examples of the use of the Report Generator can be seen in chapter 9, where most of the analysis, including the histograms was performed with it and RAQUEL.

The possibilities for the statistic analyser are numerous, building gradually a complete data analyser with powerful graphic facilities. The Report Generator may also be subject to many extensions. It could include facilities to transpose a table, substitute column values, produce subtotals and totals at the end of each page, more flexibility for defining layout formats and storing them for future use, etc.

As indicated earlier, the design and implementation of this program was motivated by both user comments and our own needs for data analysis. We were able to achieve the capabilities outlined earlier in less than a week of design, implementation and testing because we could concentrate on the user interface. All the underlying database software was provided by the existing components used to implement the query languages.

7.7 Conclusions

The software described above was designed, implemented and tested incrementally. The IQL and the software components were first tested through the PROTEUS/Monitor (section 7.5). Then RAQUEL (section 7.3) was implemented while the components were upgraded. After RAQUEL became fully relational and had been tested and demonstrated, the rest of the user interfaces were implemented.

All of the user interfaces were implemented using (re-using) some or all of the software components described in chapter 6. All access the same data, using the same name space. They provide different levels of functionality and usability to suit different needs and preferences. All the interfaces described in this chapter provide ample room for improvements and extensions and the underlying components have proved to offer an effective basis for providing rapidly and economically new interfaces.

Our experience has shown that there are two principal factors that contribute to the rapid prototyping of database systems. One is the provision of appropriate implementation tools, particularly a programming language such as PS-algol with persistence capabilities. The other factor is the availability of adequate database software components such as those proposed in chapter 6. We believe that the right choice of both significantly reduces costs and time in producing database tools, as demonstrated in this chapter.

Chapter 8

Analysis of Relational Structures Using PS-Algol

8.1 Introduction

One of the advantages of the PS-algol language (introduced in chapter 5) is that program designers are free to choose the data structures that best adapt to their applications. However, if a choice is available, then this freedom must be carefully exercised.

Different data representations may offer substantially different time and space performance. To arrive at an adequate compromise between both, the program designer has some room for calculating space utilisation but space and time performance can be most accurately determined by conducting a few experiments to measure them.

This chapter is about such an experiment. In it, an analysis of different data structures for representing tuples and relations in PS-algol is presented. However, our analysis is by no means exhaustive and there is clearly a need for more research.

Unfortunately, there is no obvious best structure. This is due to the heap orientation of PS-algol and its dynamic management of objects between main store and backing store. The programmer is restricted to manipulating objects on the heap only. He has no access to objects beyond the heap and cannot take advantage of the underlying file or buffer handling system nor any hardware characteristics [Wiederhold 77]. Futhermore he has no control over the heap management system and hence cannot anticipate the data movement between heap and disk. Variable heap size, and garbage collection further complicate the analysis.

We next give a brief introduction to some aspects of PS-algol that will help clarifying these issues. The following section contains extracts of the more detailed work presented in [Atkinson 83b, Atkinson 83a, Atkinson 83c].

8.2 PS-algol Heap Management

The base types of the language are integer, boolean, file, string and pointer. The type constructors are vector and structure. Multidimensional arrays may be formed by composing the vector construction operations. Structure classes are ordered cartesians of named fields

belonging to one of the base types, or to a vector type. Pointers are access descriptors which can reference instances of any of the structure classes but which may not reference instances of base types or vectors. That is, a pointer is not bound to a structure class. However, when a pointer is dereferenced using a structure field name, a run time check occurs to ensure the pointer is pointing at a structure with the appropriate field. The following program illustrates the use of pointers, vectors and structures:

```
! *string denotes a vector of strings

structure tuple (pntr previous, next;
                 *string columns)

! @1 of string [....] declares a string vector
! where @1 indicates the lower bound.

let first := tuple (nil,nil,
     @1 of string ["Marianita","Josefina"])
let second:= tuple (first,nil,
     @1 of string ["Matias","Magdalena"])
let last  := tuple (second,nil,
     @1 of string ["Alejandro","Francisco"])

first(next)  := second !initialize 'next' field
second(next) := last

let x := first  !pointer to first tuple
while x ~= nil do
begin
  write "'n"  ! start a new line
  for i = 1 to upb(x(columns)) do
     write x(columns,i)," "
  x := x(next)     ! point to the next tuple
end
```

Running this program yields:

```
Marianita Josefina
Matias Magdalena
Alejandro Francisco
```

All compound objects: strings, structures and vectors, are stored on the heap. There is one object identified as the *root* of the database. All objects reachable from the root are identified as persistent objects, those not reachable are transient objects. There is a garbage collector which preserves objects reachable from identifiers currently in scope.

The root of a database is made available when opening the database (a fuction call). Data may be created on the heap during a transaction or it may migrate there as a copy of some persistent data object. The second mechanism is invoked when a pointer being dereferenced is a *persistence identifier* (PID). The persistent object manager is called to locate the object and place it on the heap. The initial pointer which is a PID is that of the root object (normally an associative table) and subsequent pointers will be found in the fields of structures reached from that reference.

When a transaction is comitted, all the data on the heap that is reachable from the persistent objects during the transaction are transferred back to disk. Some anticipation of these transfers may be necessary if the heap space is insufficient for the whole transaction.

The previous issues suggest that an effective analysis should have a knowledge of the expected use of the database objects, derived from experiments. The experimental results discussed in the next sections derive from our own application's environment. The analytical analysis is more "general purpose". We have only considered string and integer attributes, booleans and reals will behave similar to integers.

The criteria for selecting a structure will be its space utilisation, the cpu time performance in dereferencing objects from the database and the simplicity of algorithms implementing operations on the structures. Space utilisation is measured in bytes, cpu time in milliseconds.

8.3 Notation

The following notation is used in the space analysis of the structures:

```
A   : Attribute
AToh: Attribute Type overhead
      strings : 4 bytes (embedded in structures)
      integers: 0 bytes
AVs: Attribute Value space utilisation
      Is : integers (4 bytes)
      Ps : pointers (4 bytes)
      Ss : strings  (if string-length <= 1
           then 0 bytes else string-length + 4 bytes)
b   : length of intermediate vectors
c   : cardinality of a relation (c is considered >= 1)
m   : number of intermediate vectors or
      intermediate lists (m*b >= c)
n   : degree of a relation
p   : number of attributes with strings of length > 1
q   : number of attributes with strings of length <=1
R   : Relation
sl  : string length
Soh: Structure overhead (8 bytes)
T   : n-ary tuple
Voh: Vector overhead (8 bytes)
```

The space used by a vector of size 'v' is 8 + 4*v bytes. The space used by a structure with no fields is 8 bytes.

8.4 Relation Space Analysis

The following analysis concentrates on structures for relations only. The space used by tuples is deferred to later sections and it is here considered as I. We do not consider the space used by possible indices over the relation.

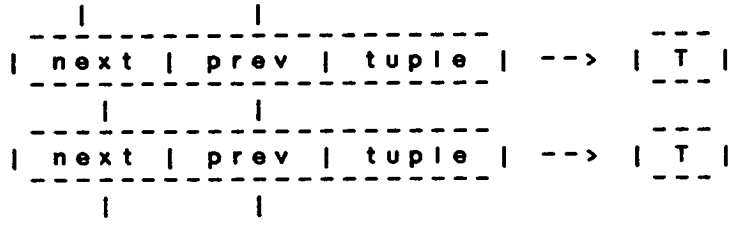
One of the requirements is that the relations ought to be traversed sequentially in both directions: starting from the first tuple and progressing a tuple at a time to the last tuple and viceversa. This is because a query evaluator or a user interface will, in general, traverse sequentially a relation e.g. a user interface displaying relations on the screen may allow the user to "scroll" the relation forwards and backwards (e.g. TABLES, see chapter 7). We therefore consider double linked lists and vector based structures only. The associative table mechanism provided in PS-algol via B-trees does not permit sequential scan and is therefore not considered. The structures analysed are:

1. Double linked list of tuples
2. Vector of Pointers
3. List of Vectors
4. Vector of Vectors

8.4.1 Relation as a Double Linked List of Tuples

Suppose a relation consists of a double linked list of *tuple-header* structures pointing to the next and previous tuple-headers and to the *tuple-body T*. The tuple pointer may be a pointer to a structure or to a vector, as shown in figure 8-1.

Figure 8-1: Relation as a Double Linked List of Tuples




```
Space used by each tuple-header:
   space used by structure (Soh) + 3 pointer = 20

Space used by Relation (c = cardinality):
   R = c * (20 + T)
```

8.4.2 Relation as a Vector of Pointers

Suppose a relation consists of a vector of pointers to tuples, as shown in figure 8-2. A vector structure offers faster access to individual tuples than a linked list but the size of the vector requires additional maintenance, especially if the relation is subject to frequent updates. An expensive but straightforward implementation would increase or shrink the vector, element by element, as required. A more sophisticated approach would parameterise the changes to the size of a vector e.g. an intial vector size which, once filled, is extended by a percentage of its current size. Space used by Relation = space used by vector + space used by tuples

$$R = (8 + 4*c) + c*T = 8 + c*(4 + T)$$

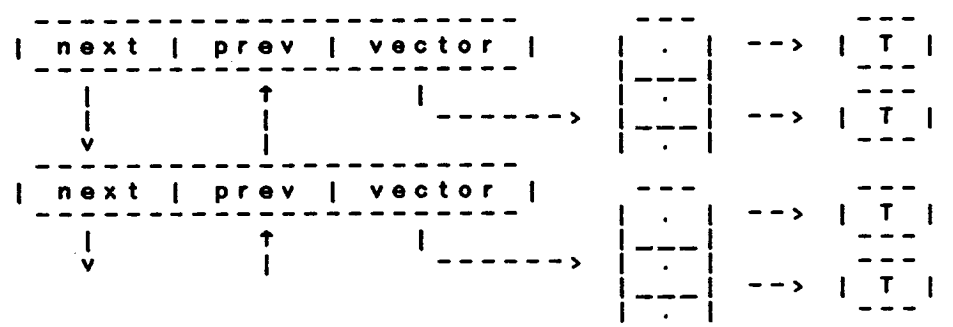
Note that a vector of strings (all attributes compacted in one string) use the same amount of space.

Figure 8-2: Relation as a Vector of Pointers to Tuples

8.4.3 Relation as a List of Vectors

To assist in the task of managing vector sizes, especially for large relations, where a single vector may be expensive to copy when extended or even impossible to handle due to the heap size limitations, a combination of the above two structures appears appropriate. Suppose a relation consists of a double linked list of vectors pointing to tuple bodies, as shown in figure 8-3.

Figure 8-3: Relation as a Double List of Vectors



Assume that there are 'm' list elements and therefore also 'm' vectors with tuple pointers. To simplify the analysis assume also that all vectors are complete with 'b' tuple pointers, hence:

c = m*b

In practice and providing that relations are kept as compact as possible, most vectors (excepting probably the last one) would be complete. If the relation is large and the size of the vectors is relatively small, then the space analysis will closely reflect the practical situation.

Space used by Relation:

R = space used by 'm' lists + space used by
'm' vectors + space used by 'c' tuples

Space used by 'm' list elements
= m*(space used by list structure + 3 pointers)
= m*(8 + 3*4)=m*20

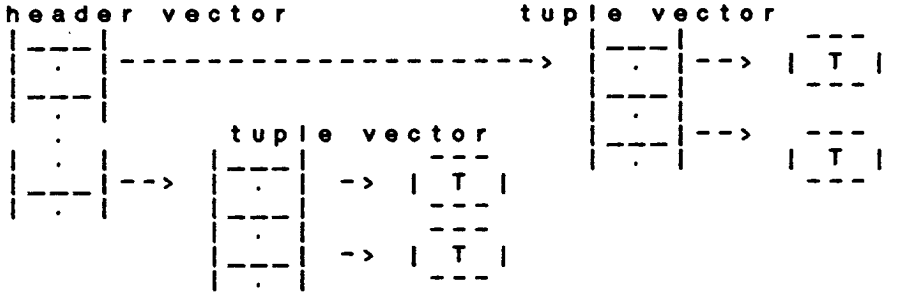
```
Space used by 'm' vectors of size 'b' = m*(8 + 4*b)
Space used by 'c' tuples : c*T

R = m*20 + m*(8 + 4*b) + c*T
R = 28*m + c*(4 + T)
```

8.4.4 Relation as a Vector of Vectors

Another possibility of handling large relations is to use matrices which can also be seen as a tree of vectors, with the last vectors containing references to the tuples. Assume that a relation consists of a *header vector* pointing to *tuple vectors*, as shown in figure 8-4. The analysis of further levels of indirection to tuples (a tree of vectors) is similar and is here restricted to two levels.

Assume that the header vector has size 'm' and that all tuple vectors contain the same number of tuples (equal to their size=b). Hence, the cardinality c = m*b

Figure 8-4: Relation as a Vector of Vectors




```
Space used by:
  1   header vector of size 'm' = 8 + 4*m
'm' tuple vectors of size 'b' = m*(8 + 4*b)
'c' tuples = c*T
Space used by relation:

R=8 + 4*m + m*(8+ 4*b) + c*T = 8 + 12*m + c*(4+T)
```

8.5 Comparison of Relation Structures

A summary of the space cost equation for each Relation is: (Consider all relations of cardinality c >= 1)

```
Double Linked List Of Tuples:          c*(20 + T)      (1)

Vector Of Pointers:                 8 + c*(4 + T)      (2)

List of Vectors:                 28*m + c*(4 + T)      (3)
(m = Number of vectors and list elements)

Vector of Vectors:        8 + 12*m + c*(4 + T)      (4)
(m = Number of tuple vectors)
```

The best structure in terms of space efficiency is (2) followed by (4), except in the unusual case of singleton relations (c=1) where (1) takes the second place.

All structures have the same factor c*T which is the space used by all tuples in the relation. Without it, the relation structures can be compared in terms of the space overhead they introduce.

Considering that the effectiveness of each structure is best reflected in large relations, we can simplify the analysis by assuming that a large relation has a cardinality much greater than the number of tuple vectors (case (3) and (4)), thus:

$$4*c >> 28*m \quad => \quad c >> 7*m$$

In this case, the constants and the 'm' factor are negligible. The equations for the overhead of relation structures become:

$$(1) = 20*c$$
$$(2) = (3) = (4) = 4*c$$

8.6 Tuple Space Analysis

There exist many candidates for tuple structures and again, the choice is not obvious. We are interested in the compactness of tuple structures and in referencing attributes individually. Attribute values may all be stored as strings, providing a uniform management but introducing some extra cost involved in the type translation. String attribute values may be compacted in one string, but operations using vertical partitions of a relation (e.g. project, grouping functions, joins), may have to copy substrings of the tuple, generating new objects on the heap. If the tuple is a vector then these operations would only create references to the elements of the vectors they are interested in.

The following PS-algol structures are used to hold attribute values in some of the cases analysed below.

```
structure sa (string sa.s)                    ! atom
structure ia (int    ia.i)                    ! atom
structure se (string se.s; pntr se.p)! list element
structure ie (int    ie.e; pntr ie.p)! list element
```

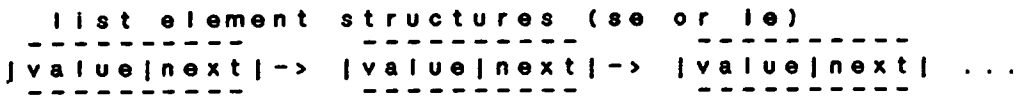
The analysis includes list of structures, vector of structures and vector of strings, strings and records.

8.6.1 Tuple as a List of Attributes

Assume a tuple is formed using List Element structures, as shown in figure 8-5. Space used by each attribute = A_i = space used by structure (Soh) + Attribute Type (AToh=4 for strings, AToh=0 for integers) + Attribute Value(AVs=sl+4 for strings, AVs=4 for integers) + pointers (Ps)

$$A_i = Soh + (AToh + AVs) + Ps = 12 + (Atoh + Avs)$$

Figure 8-5: Tuple as List of Attributes

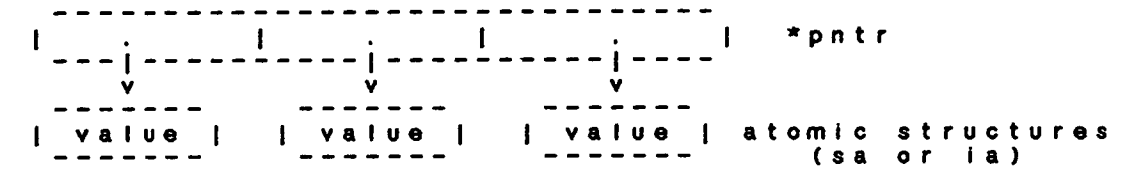


Space used by n-ary tuple: $T = 12*n + \Sigma_1^n(AToh_i + AVs_i)$

8.6.2 Tuple as a Vector of Pointers

Assume that a tuple consists of a vector of pointers pointing to atomic structures, each containing an attribute value, as shown in figure 8-6.

Figure 8-6: Tuple as a Vector of Pointers



Space used by each attribute:

$A_i = Soh + AToh + AVs = 8 + AToh + AVs$

Space used by vector of size 'n' = $Voh + Ps*n = 8 + 4*n$

Space used by n-ary tuple:

$T = 8 + 4*n + 8*n + \Sigma_1^n(AToh_i + AVs_i)$

$T = 8 + 12*n + \Sigma_1^n(AToh_i + AVs_i)$

8.6.3 Tuple as a Vector of Strings

The previous two structures may keep the attributes in their original type (e.g. integer, boolean, etc.). For this structure, assume that all attribute values are of type string or have been converted to a string. A tuple is contained in a vector of strings with each attribute in a vector element, as shown in figure 8-7.

Figure 8-7: Tuple as a Vector of Strings

```
Consider    p attributes with strings of length > 1
and         q attributes with strings of length <= 1
The degree of the relation is n = p+q

Space used by each 'p' attribute= sl_i+Atoh+Ss=sl_i+4+4
Space used by each 'q' attribute (Ss=0)=Atoh=4
Space used by n-ary tuple:
```

$$T = Voh + \Sigma_1^p(sl_i) + p*8 + q*4$$

$$T = 8 + \Sigma_1^p(sl_i) + 4*(p+q) + 4*p$$

$$= 8 + \Sigma_1^p(sl_i) + 4*n + 4*p$$

```
if all attributes have length > 1 then
```

$$T = 8 + \Sigma_1^n(sl_i) + 8*n$$

It is also possible to encode similarly a relation having only integer attributes. We have not explored that possibility nor the frequency this happens and what procedure to follow if a string attribute is introduced.

8.6.4 Tuple as a String

This structure offers a very compact way of packaging string attributes into a single string. A potential disadvantage lies in the handling of this structure in operations using vertical subsets of the relation (project, join and group operations) which would need to copy substrings of this string. Another disadvantage is a slower access to individual attributes as compared to a vector of strings. One of its advantages is to diminish the space waste of attribute values that are not modulus 4 by compacting them in one string. It is now the whole tuple which is extended, if necessary, to modulus 4 length.

Assume all attribute values are strings, separated by a special character. If the tuple has only one attribute then it contains no separator. Additional attributes are prefixed with the separator, as shown in figure 8-8.

Figure 8-8: Tuple as a String

```
 ---------
| Mariana | 1-ary tuple
 ---------
 --------------------
| Mariana|Josefina   |
 --------------------
2-ary tuple: att. are separated by character '|'
```

```
Space used by n-ary tuple:
if
 n=1 and sl_1 <= 1          !null or 1 char. string
or
 n=2 and sl_1=0 and sl_2=0  !separator only
then T=Atoh=4
else T=Atoh+ Σ_1^n(sl_i)+4+n-1     !n-1 separators
          T=Σ_1^n(sl_i)+7+n
```

There are other possibilities for further compressing data. For example, storing integers as base 128 will often encode in one or two bytes.

8.6.5 Tuple as a Record

Ideally, a tuple would be encoded as a PS-algol structure (a record), holding the attributes in its fields, as in the example of figure 8-9.

Figure 8-9: Tuple as a Record

```
structure person (string name, address;
                  int    phone;
                  bool   married)
```

This structure is presently not implementable in PS-algol, because structure class definition may not be updated at run time, that is, they are not first class objects e.g. adding a new attribute to the "person" class requires recompilation. It is included here for comparison purposes only.

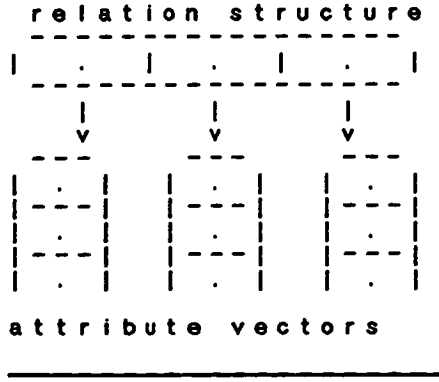
Space used by n-ary tuple = space used by structure + $\Sigma_1^n(AToh_i + AVs_i)$

$T = 8 + \Sigma_1^n(AToh_i + AVs_i)$

8.6.6 Transposed Relation

In a transposed relation, a vector (or a vector of vectors) is used for each attribute. The relation structure may consist of a vector itself, a list or a record, as indicated previously. This situation is shown in figure 8-10. This structure offers fast access to individual attribute values but is potentially inefficient for operations using horizontal partitions of a relation (e.g. the select, join and set operations), which would have to copy portions of these attribute vectors into other vectors, as well as for update operations where the extending or shrinking of the vectors may become too expensive. It is therefore not further considered.

Figure 8-10: Transposed Relation



8.7 Summary of Tuple Structure Equations

A summary of the space equations for the tuple structures that will be further analysed is: (we assume strings of length sl > 1 and degree n > 1)

```
Attribute List       =        12*n  + Σ_1^n(AToh_i + AVs_i)
Vector of Pointers   = 8  +   12*n  + Σ_1^n(AToh_i + AVs_i)
Vector of strings    = 8  +    8*n  + Σ_1^n(sl_i)
String (Compact)     = 7  +      n  + Σ_1^n(sl_i)
```

8.8 Space used by Relations and Tuples

The following list is a combination of relation/tuple structures. Large relations are assumed (c >> 7*m), where 'c' is the cardinality of the relation and 'm' is the number of vectors in the combined List/Vector structures.

Vp is a vector of pointers and Vs is a vector of strings in a structure. To simplify the analysis, it is assumed that all strings have same length 'sl'.

```
List/List     = c*(20  +  12*n  + Σ_1^n(AToh_i + AVs_i))
List/Vp       = c*(28  +  12*n  + Σ_1^n(AToh_i + AVs_i))
List/Vs       = c*(28  +   8*n  + n*sl)
List/String   = c*(27  +     n  + n*sl)
Vector/List   = c*( 4  +  12*n  + Σ_1^n(AToh_i + AVs_i))
Vector/Vp     = c*(12  +  12*n  + Σ_1^n(AToh_i + AVs_i))
Vector/Vs     = c*(12  +   8*n  + n*sl)
Vector/String = c*(11  +     n  + n*sl)
```

8.9 Case Analysis

To compare the structures, the space used by each of them in relations containing either integers or strings is measured. Note that reals and booleans would behave much like integers and that date and time where supported, would probably encode to integers or strings.

8.9.1 String Attributes

All the space equations defined for relations and tuples in the previous section contain the factor c. If all attributes have the same length sl>1 then the factors c and n*sl, common to all structures, can be ignored. What remains is the overhead in bytes introduced by each structure per tuple, in terms of the degree 'n'. (Σ_1^n(AToh$_i$ + AVs$_i$) = 4*n + n*(sl+4) = 8*n + n*sl)

```
List/List     = 20  + 20*n
List/Vp       = 28  + 20*n
List/Vs       = 28  +  8*n
List/String   = 27  +    n
Vector/List   =  4  + 20*n
Vector/Vp     = 12  + 20*n
Vector/Vs     = 12  +  8*n
Vector/String = 11  +    n
```

The above list shows that the combinations Vector/String and List/String have the best performance, except for degree n < 3 where Vector/Vs is second.

8.9.2 Integer Attributes

We assume that all attributes are of type integer and to simplify the analysis, all have the same number of digits: case a) all have 3 digits and case b) all have 6 digits. Case a) corresponds to the average number of digits found in our own applications. Case b) is provided for comparison. The additional space used by structures converting integers to strings values for strings is shown in the next table ($\Sigma_1^n(AToh_i + AVs_i) = 4*n$).

Relation/Tuple	sl = 3	sl = 6
List/List	=20 + 12*n + 4*n	20 + 16*n	20 + 16*n
List/Vp	=28 + 12*n + 4*n	28 + 16*n	28 + 16*n
List/Vs	=28 + 8*n + n*sl	28 + 11*n	28 + 14*n
List/String	=27 + n + n*sl	27 + 4*n	25 + 7*n
Vector/List	= 4 + 12*n + 4*n	4 + 16*n	4 + 16*n
Vector/Vp	=12 + 12*n + 4*n	12 + 16*n	12 + 16*n
Vector/Vs	=12 + 8*n + n*sl	12 + 11*n	12 + 14*n
Vector/String	=11 + n + n*sl	11 + 4*n	11 + 7*n

The table indicates that structures storing values as strings compare well with the others. The graph in figure 8-11 shows the performance of four structures for case a) in relations with degree from 1 to 20.

8.9.3 Effect on Relational Operations

To find the effect on the space utilisation under the worst, but very common case of a query that projects all attributes of a relation, the following table shows the additional space created by the projection operation for each relation structure.

The tuple structures List, Vector of Pointers (Vp) and Vector of Strings (Vs) create structures containing pointers to the argument relation. The String tuple structures copy the entire relation. To simplify the analysis all strings have the same length sl > 1.

Operation: R2 := R1 projected on all attributes.

List/List	= c*(20 + 12*n + 4*n)	= c*(20 + 16*n)
List/Vp	= c*(28 + 12*n + 4*n)	= c*(28 + 16*n)
List/Vs	= c*(28 + 8*n)	
List/String	= copy	= c*(27 + n + n*sl)
Vector/List	= c*(4 + 12*n + 4*n)	= c*(4 + 16*n)
Vector/Vp	= c*(12 + 8*n + 4*n)	= c*(12 + 12*n)
Vector/Vs	= c*(12 + 8*n)	
Vector/String	= copy	= c*(11 + n + n*sl)

Comparing the cheapest copy with Vector/Vs

$$\text{Vector/String} < \text{Vector/Vs}$$

$$11 + n + n*sl < 12 + 8*n$$

$$sl < \frac{1 + 8*n}{n}$$

Vector/String is always better if strings <=8 char.

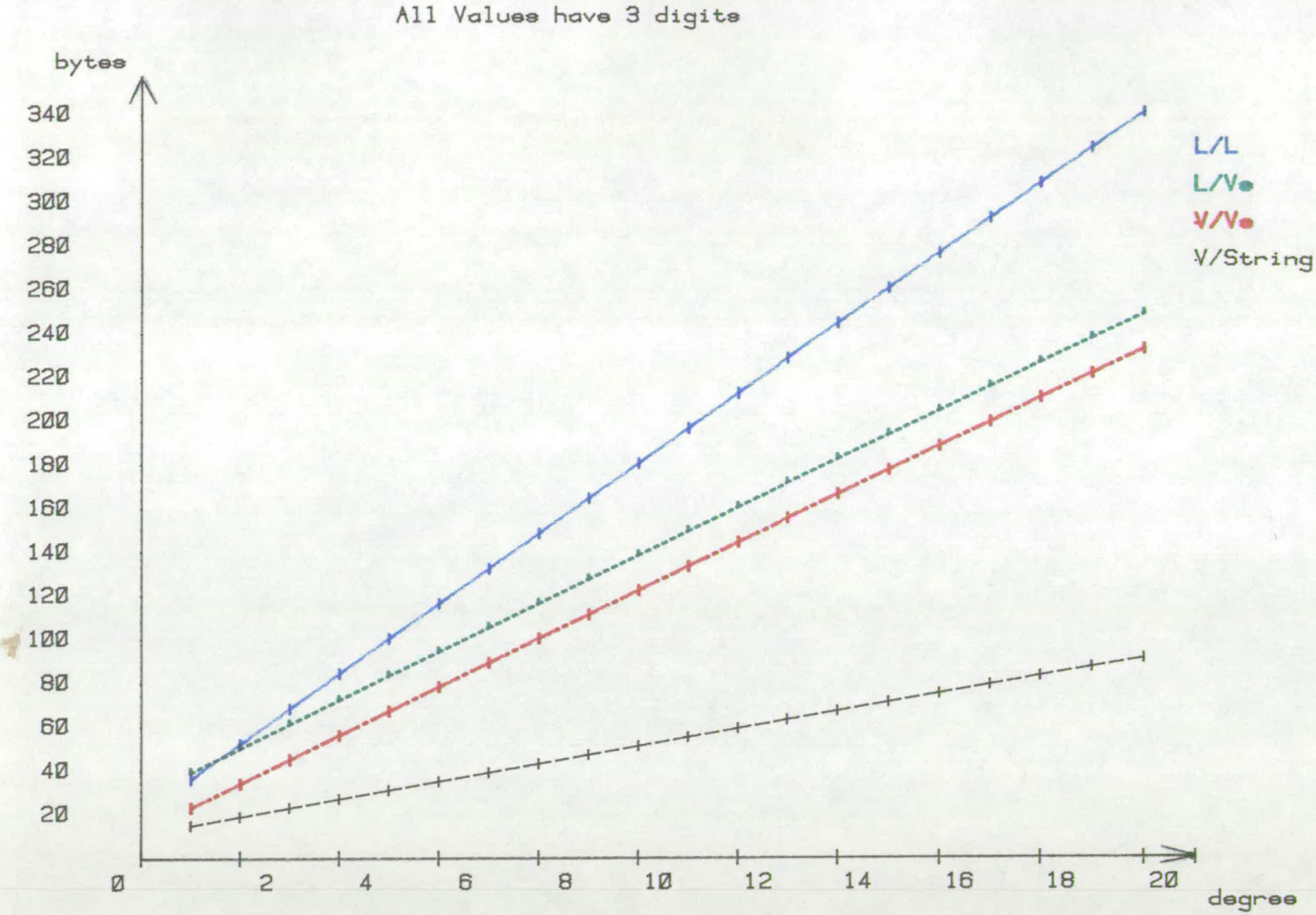


Figure 8-11: Space used by Integer Attributes

8.9.4 Conclusions

The above analysis suggests the use of compacted structures with all attributes stored as strings if these are relatively small, or if the relation has only integer attributes. If this is not the case and relations are used to contain textual data such as addresses, documentation, etc., then vectors of strings as tuple structures would perform better.

8.10 Programmability

Programmability or ease-of-programming is one of the most tempting factors when choosing data structures, particularly if the system is a prototype where functionality and rapid programming is favoured against time performance and space utilisation. However, in interactive systems the responsiveness of the user interface is particularly important, and may be decisive when assessing the usability of the system. In such cases, a prototype for an interactive system must consider, in as much as possible, those structures that will provide a more satisfactory time performance.

8.10.1 Tuple Structures

If a tuple is a linked list of attribute structures with attributes of different type, then dereferencing attributes may involve substantial processing. The traversing algorithm must consult the type of the structure containing the attribute before moving to the next attribute. In contrast, dereferencing an attribute in a vector of attributes is simple via indexing. Traversing tuples is used in every relational operation or when displaying or "browsing" through a relation. In addition to the traversing problem, the list of attributes structure proved to be the most inefficient in terms of space utilisation.

If a tuple is a vector then it is faster to traverse than a list or a string of attributes. The problem which remains is the maintenance of the vector. Extending (adding an attribute to the tuple) or shrinking the tuple is potentially inefficient. This is investigated in the next section where it is concluded that the system is relatively efficient in dealing with vectors. A vector oriented system may implement a few functions to copy, extend and shrink vectors making the rest of the operations rely on their performance (see section 6.9, chapter 6). This has been tested by experiment resulting in cleaner and more compact algorithms compared to those for maintaining lists.

If a tuple is a string, with attributes separated by a special character, the structure becomes very compact at the expense of additional processing for dereferencing the attributes. However, once a few functions to dereference the attributes are implemented the only question which remains is whether the processing overhead in this task outweights the advantage of compactness.

As the previous section has shown, the storing of attributes in individual structures is the most inefficient choice. It also makes programming much simpler at the storage level to deal with only one type of data. Considering a query evaluator where the most frequent

operations are probably projection and selection using equality tests (see chapter 9), type conversion will be rarely needed. Moreover, being type conversion a relatively fast operation, its impact is negligible.

In summary, all structures require a set of housekeeping functions for their maintenace on which the rest of the operations will rely. The simplicity of having one type of data (strings) greatly simplifies the programming at storage and query processing levels. Considering these arguments plus the results of the space utilisation analysis, vectors of strings and strings emerge as the best candidates for tuple structures.

8.10.2 Relation Structures

In considering structures for relations, the size of vectors becomes a major issue due to its maintenace cost. On the other hand, the present PS-algol version restricts the maximum size of vectors to around 15000 elements. Therefore, groups of vectors (list of vectors or vector of vectors), containing references to the tuples, are considered in the following analysis.

Small vectors are fast to extend or shrink, thus more easy to maintain but too many small vectors requires more effort in switching from one group of vectors to the other and has a greater space overhead.

To ease the maintenance of a relation, its vectors may provide more space than actually needed. In an environment where updates are a common operation this situation will require additional effort to keep the vectors of a size related to the number of tuples they contain.

In our applications, the cardinality of the relations tend to be in the range of 100 to 5000 with the average closer to the lower end. A vector size of 1000 will leave the majority of the relations with only one vector during their entire life time, making extensions a rare operation. Groups of vectors may be thus arranged in a short double linked list or in a vector of vectors.

The other candidate for a relation structure is the double linked list of tuples. Its advantage is that it produces no waste of space due to unused list elements. Tuples are inserted and deleted by using simple list maintenance algorithms. On the other hand, the additional space introduced by the list pointers ('previous' and 'next' tuple pointers) makes the potential waste of space introduced by vectors less unattractive. Traversing a list is less efficient than traversing a vector. This may be important if scanning a relation in search for a tuple becomes a relatively frequent operation.

In summary, a double linked list of vectors or a vector of vectors appears to be the best candidate for a relation structure.

8.11 Cpu Time Analysis

In this section, the cpu time performance of the four Relation/Tuple structures that offer the best space utilisation is measured. All structures have all attributes of type string. The operation measured is the traversal of the entire relation, dereferencing all attributes by assigning them to a string variable. The objective is to determine the processing overhead of dereferencing attributes compacted in a string as compared to indexing an attribute from a vector of strings.

The structures analysed are:

1. List/String: A double linked list as the relation structure and a string (all attributes compacted in a string and separated by a special character) as the tuple structure.

2. List/Vector of Strings: A double linked list as the relation structure and a vector of strings as the tuple structure.

3. Vector/Vector of Strings: A matrix of strings as the relation and tuple structure: (m := vector 1::cardinality, 1::degree of strings).

4. Vector/String: The relation is a single vector of strings with each tuple as a compacted string.

Each structure combination is stored in a database containing only one relation with the required number of attributes and tuples. Once the database has been created, a program retrieves the whole relation twice. This duplicate retrieval is to measure the effect of the second pass after objects have been imported into the heap.

Relations have been organised in two groups. Within each group, all relations contain the same type and amount of information for all structure combinations.

The first group consists of relations of degree 2 and cardinality 3000 with all attributes containing 3 bytes each. This corresponds to average values for degree and integer size in our applications where various binary relations are used to interconnect the information from other relations. (see figure 4-1 and 4-2 in chapter 4 where the binary relations "Attend", "Organise", Workfor" and Belong" contain integer attributes).

The second group consists of relations of degree 5 and cardinality 2000 with all attributes containing 10 bytes each. These numbers have been chosen to compare with the first group and to fit the storage limitations of the departmental computer.

In the first pass, when data is imported from the database (figures 8-12 and 8-13), the Vector/Vector of Strings structure performs consistently better. The List/Vector of Strings becomes non-linear after retrieving 400 tuples in one case and 1700 in the other.

Once the data has been brought into the heap and the attributes are dereferenced for the

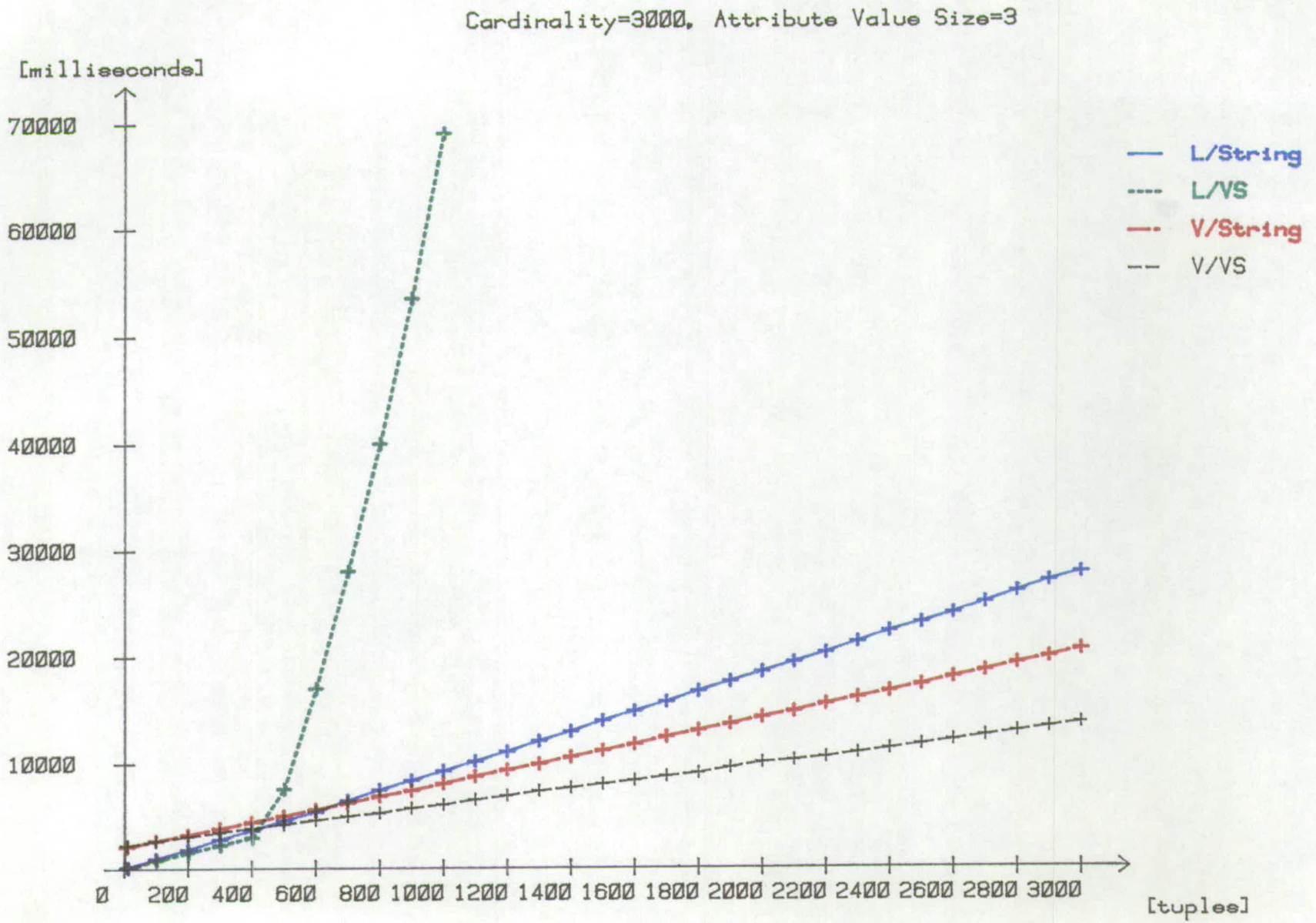


Figure 8-12: Cost in Traversing a Relation: Degree=2, First Pass

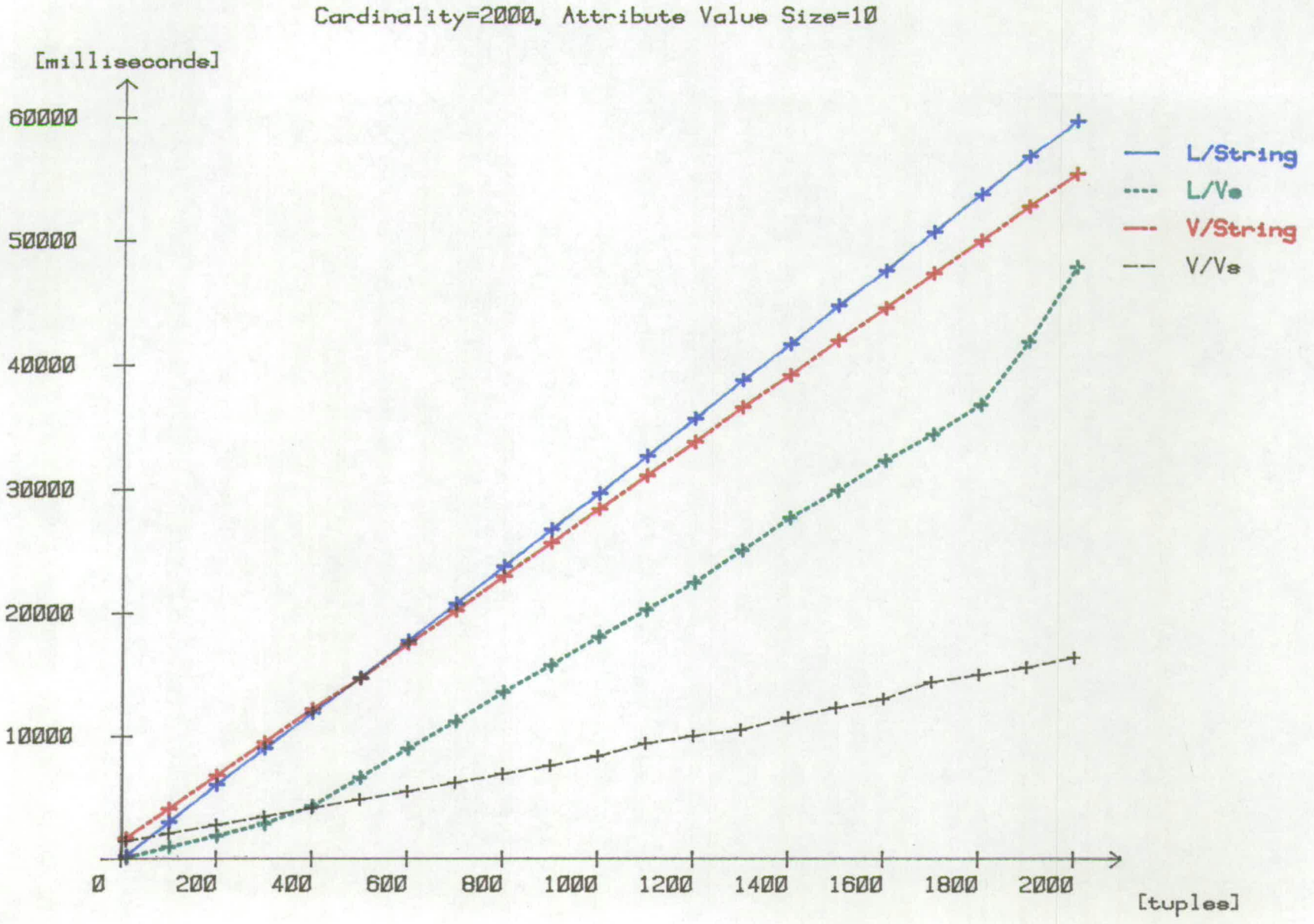


Figure 8-13: Cost in Traversing a Relation: Degree=5, First Pass

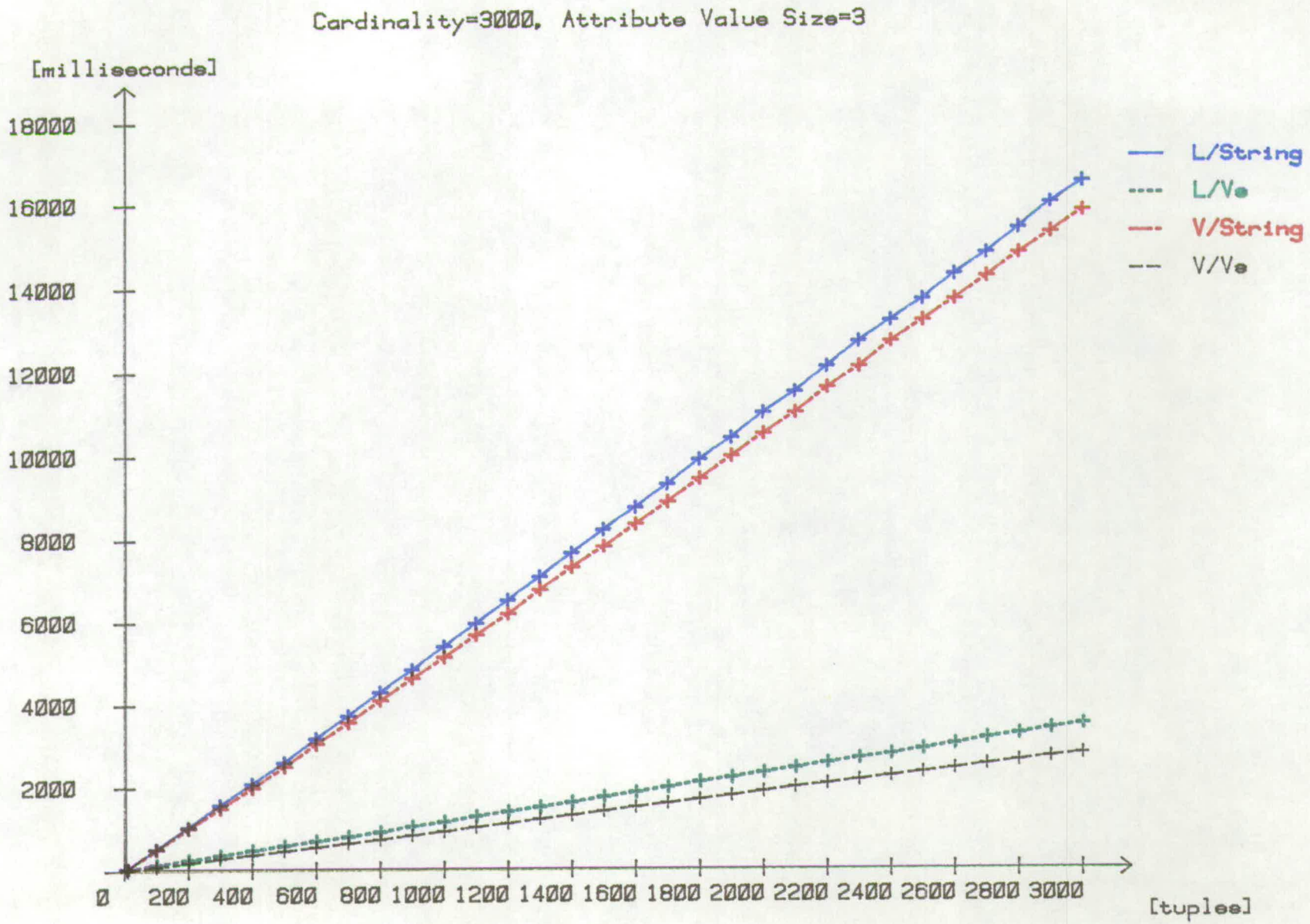


Figure 8-14: Cost in Traversing a Relation: Degree=2, Second Pass

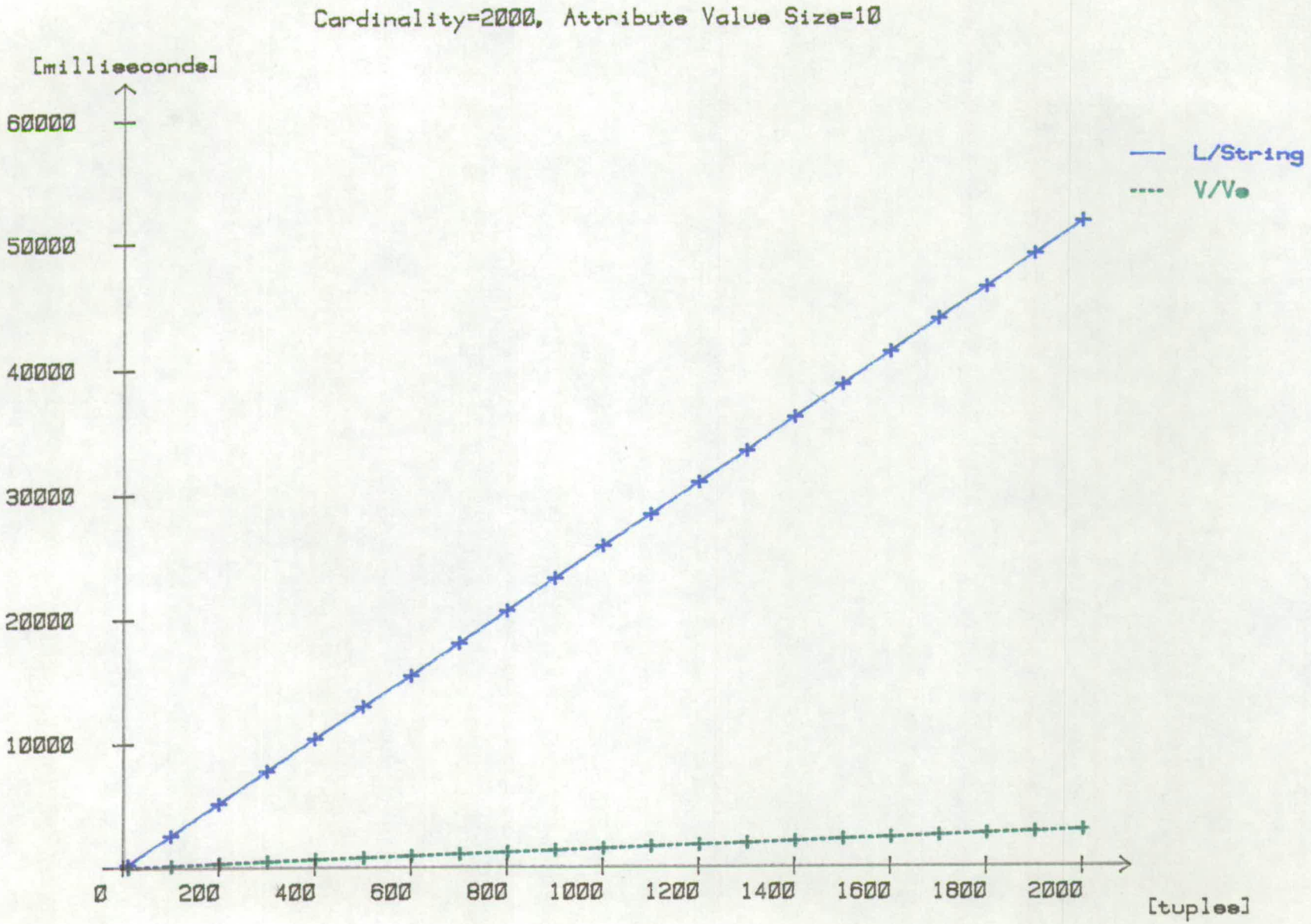


Figure 8-15: Cost in Traversing a Relation: Degree=5, Second Pass

second time (figures 8-14 and 8-15), the Vector of Strings tuple structures win over the two others. This is simply the overhead in dereferencing an attribute value in a vector requires less cpu time than scanning a string to find attribute separators, calculating the length of the value. Figure 8-15 contains only the List/String (Vector/String is similar to it) and Vector/Vector of Strings structures (List/Vector of Strings is similar to it).

The best choice is clearly the Vector/Vector of Strings structure, otherwise it is a case of space versus time and of single-use versus repeated-use of the data.

8.12 Conclusions

Before drawing general conclusions, we note that our results should be regarded only as preliminary since there is a clear need for more investigation.

Considering the above discussion on space and cpu performance and programmability, the decision of which structure combination to adopt relies on the characteristics of the application.

If space usage is critical and relations tend to contain textual data, then the Relation/Tuple structure combination Vector/String would be preferred. The Vector of Strings tuple structure appears simpler to use with better performance with re-used data. If relations are small, then List/Vector of Strings may be preferred due to its inexpensive maintenance.

However, relations are expected to become larger with time and it was considered as important to have a "smooth" transition from small to large volumes of data. Vector/Vector of Strings demonstrated consistently better (linear) cpu performance than all other structures. It is also the best of the non-compact structures in terms of space utilisation and also the simplest to traverse but requires some maintenance with respect to relation vector extension and shrinking. In general, the Vector/Vector of Strings combination seems to be the best choice.

Chapter 9

Recording Query Language Usage to Improve System Usability

9.1 Introduction

The present trend of using computers in a wide range of applications by an increasingly large number of non-experts is modifying the attitude of software designers toward a more careful consideration of the user interfaces.

Traditionally, it has been users who had to adjust to the applications software. After all, the main direct users were computer professionals. A typical example is that of the industrial Data Processing Department where a group of programmers and system analysts act as mediators between users (e.g. managers) and computers.

This trend is rapidly reversing. Software (and hardware) manufacturers have realised that a potentially huge market for computer applications in almost every human activity is ready (and eager) to use computer technology. However, as any other sophisticated technology, a computer product must be first properly 'human' engineered to make it attractive to the non-expert users [Cuff 79]. From an economic point of view it has been reported that

> *"for over 95 per cent of the human/machine interactions, people costs are greater than machine costs. Actions that reduce the human costs and simplify the human interface will have the greatest impact on the growth of computer industry"* [Benbasat 81].

A pedagogical and psychological ingredient had to be incorporated in the design of user interfaces so that applications are now built around a model of user attitudes and motivations toward computer technology. This is reflected in the publicity of commercial products where the emphasis has shifted from the technicalities of a system such as 'mips' or 'cpu clock rate', to the so-called ergonomic aspects such as friendliness and ease of use.

User interfaces are not only designed to provide the right functionality but also the right motivations and feedback.

> *"User attitudes can dramatically affect the learning and performance with interactive systems"* [Shneiderman 79].

User feelings are considered in designing system interfaces. Shneiderman provides many examples:

> *"in designing a system for novices, every attempt should be made to make the user at ease, without being patronising or too obvious"* [Shneiderman 79].

Various experiments in the database area have been conducted to compare two or more query languages: ease of use, ease of learning, procedurality versus non-procedurality, etc. have been the primary focus of these research [Thomas 75, Reisner 81, Moran 81, Welty 81].

These experiments were in general designed and conducted as laboratory research. Query language, users, teaching method, testing procedures, etc were carefully controlled in order to obtain statistically significant results.

These laboratory tests may, however, suffer from their very nature in that the real world user may learn and use a tool in a way which is different from that artificially set up by the experimenters. For example, learning a query language with an on-line terminal, with the system providing immediate feedback in the form of query results or error messages, can be completely different from the class room method using overhead projectors or blackboard.

Although very important as a method for understanding user behavior, these experiments should be complemented with 'realworld' observations where the user environment is not disturbed any more than necessary.

Most of these experiments concentrate on the analysis of cognitive factors in language issues, thus isolating the language from the rest of the system. There is however, another important motivation for analysing user behavior in database applications which also positively affects the man-machine interaction. By studying query/update patterns a system can be tuned towards a more efficient processing of these requests. This is particularly important in interactive query languages where the speed of the system is a prime factor in its usability. Optimization parameters (e.g. indices) may be automatically derived from the continuous observation and analysis of user queries.

Ideally, a database application becomes a dynamic system. It evolves with its users, by getting to know them through studying the way it is used. A system improves its effectiveness as a tool by adapting itself to its users. This is essentially the proposal made by Stocker and Dearnley [Stocker 73, Dearnley 74].

In a multi-language environment such as the one we propose, the database system in general and the user interfaces in particular are implemented initially according to the designer's intuition of what is convenient for the users. This intuition should be then continuously improved by recording and analysing user data.

The research reported in this chapter is based on the 'realworld' approach. The database query and update language Edquse [Wilson 81], which is used in the computer science department by a variety of users, was modified to gather several language usage parameters. The objective was to identify those properties of the data that could be used to improve the language and the underlying database system. It was also intended to observe the degree of complexity of the queries issued i.e. number of operations involved in a query. The

suitability of the language to support complex processing requirements can be inferred from the degree of complexity of the applications.

The use of language commands, the structure of queries, query syntax errors, query evaluation data, etc. have been gathered since November 1981.

We also present in a final section, some preliminary results of the use of the Raquel language, which has been used by third year undergraduate and MSc students for about six months since January 1983. The next sections deal exclusively with Edquse data.

9.2 The Experiment

The analysis presented in the next sections is by no means exhaustive, but it is noted that the framework for a more extensive statistical analysis is set up. It is also noted that most of the analysis was carried out using the project's user interfaces RAQUEL and the Report Generator (see chapter 7).

9.2.1 The Language

The query language Edquse [Wilson 81] was briefly introduced in chapter 2 and compared with TABLES in chapter 7. It is an interactive data base system with facilities for querying, updating and manipulating a database. It is based on the relational database model [Date 81] with a screen oriented query handler similar to QBE [Zloof 75] and having therefore the same level of procedurality [Date 81].

Edquse models applications as a collection of relations. Operations on relations are mapped into relational operations which are applied to the stored data.

Edquse has a two dimensional syntax, as opposed to left-right, top-bottom syntax of SQL (see chapter 2, figures 2-5 and 2-6) An Edquse query is formulated by placing elements into skeleton tables which have been displayed on a screen. The result of a query is displayed as a relation on the screen. A few commands allow browsing through the result relation. Updates are achieved by using a simple screen editor on a result relation.

9.2.2 The Users

Edquse users can be broadly grouped into *expert* and *casual* users. The first group comprises 2 members of staff who use Edquse regularly for course administration. The second group consists of 163 users, mainly first and third year computer science students.

Students normally have their first encounter with Edquse in one or two one-hour lectures where blackboard, visual aids such as overhead projectors and 'paper and pencil' exercises are used to introduce the language to them. These lectures may be followed by a 20 minute demonstration of the language at a terminal. They are then given an exercise they have to

complete by using Edquse at a terminal. Further learning of the language is left to themselves.

First year undergraduates use Edquse on an ICL/EMAS system, third year students on VAX/VMS.

Other users are Msc students, members of staff (e.g. course organisers), visitors, secretaries, and industry professionals participating in software courses. All of them use the VAX/VMS system. The learning process of these users (carried on through 'corridor conversations', computer mail to an Edquse 'expert', short personal demonstration, etc) is in general very different from the reported "teach the language, then use it" method.

The following analysis concentrates mainly on the *casual* user group. We indicate where relevant, differences with the *experts* and between the ICL/EMAS and VAX/VMS use.

The following notation is used:

EMAS: Edquse users working with ICL/EMAS system.
VAX : Edquse users working with VAX/VMS system.
experts: the 2 "regular" Edquse users
casual : all non regular users
total : casual + experts.
Std dev: standard deviation

9.3 Data Analysis

The data collected can be classified under the following categories:

1. Language usage: number and type of language commands used throughout a session.

2. General sessions, queries and users statistics.

3. System performance: cpu time required to process a query.

4. Result relation: cardinality and degree of result relation.

5. Query errors: those reported by the syntax analyser.

6. Query structure: the type of relational operations and their frequency of use in a query. It is noted that Edquse implements three relational operations: project, select and join. Every Edquse query contains one and only one projection.

9.4 Language Usage

Edquse works in three modes: Command, Show and Update. Transfer between modes is implied by the use of a particular command, or by using a command associated with another mode.

1. **Command** mode is entered at the beginning of a session. Queries can be formed in it, results can be transferred to external files, new relations or columns can be added, etc.

2. **Show** mode is entered after the evaluation of a query has finished and the result has been displayed on the terminal. The result relation can be then traversed using cursor positioning keys.

3. **Update** mode is entered after processing a query for update. In this mode, a simple editor is used to insert tuples, delete tuples, insert columns and exchange column values. An update can be abandoned, leaving the database in the state it was before the update, or committed in which case the updates are effected in the database.

The following sections provide the results of the Analysis of the use of the Edquse language, mainly by the *casual* users. The word *table* is sometimes used instead of *relation*.

Figure 9-1: Language Usage Totals

(*casual* users only)

Total number of commands	: 62,390
Total number of sessions	: 1,070
Average number of commands/session	: 58.3

9.4.1 Command Mode Analysis

The total Number of "Command Mode" commands is 50,736 (81.3 % of total). A summary of the results is the following: The relatively high use of the **tables** command, used to obtain the list of existing relations (tables), suggests the need for more meta-data information such as a **columns** relation. The sum of the **run** and **update** commands includes the queries producing syntax errors.

The commands used for qualifying tuples (constant, =, #, <, <=, > and >=) reflect a bias toward equality (constant and = have the same effect). From the 3362 tuple qualification commands, 3189 (94.8 %) were used for equality tests. This supports the need for indices on the columns more frequently referenced.

Figure 9-2 indicates that 80.4 % of the main commands were used for preparing queries or updates. These commands are: get table, up, down, left, right, next, print, example,

Figure 9-2: "Command" Mode Commands

command name	times used	%	% total	comment
tables	1055	2.1	1.7	show database table names
get	9655	19.0	15.5	display table skeleton on
run	4470	8.8	7.2	compile query, if no errors then evaluate it, display result table on screen and enter show mode
forget	3709	7.3	5.9	exclude table from query
update	373	0.7	0.6	as "run", entering upd. mode
up	360	0.7	0.6	move cur.1 line up, if any
down	469	0.9	0.8	move cur.1 line down, if any
left	2301	4.5	3.7	move cur.1 col.left, if any
right	7958	15.7	12.8	move cur.1 col.right, if any
next	72	0.1	0.1	move to next element in col
print	10018	19.7	16.1	put "p." element to include column in result table
kill	2137	4.2	3.4	rub out column element
example	4500	8.9	7.2	put example defining join with another table
constant	373	0.7	0.6	put constant defining select operation ('=' test)
=	2816	5.6	4.5	same as constant
#	4	0.0	0.0	define inequality
<	24	0.1	0.0	
<=	7	0.0	0.0	
>	108	0.2	0.2	
>=	30	0.1	0.1	
add	251	.0.5	0.4	define table and its cols. (bulk load may follow)
zap	46	0.1	0.1	delete table from database

constant, kill, #, =, <, <=, <, >=. It would be interesting to analyse the time spent by a user defining a query and how this correlates with the correctness and complexity of the query. Better help facilities, more guidance or a Bachman diagram may improve the "thinking time".

9.4.2 Show Mode Analysis

The Show mode is entered after a relation has been displayed on the terminal screen. The screen can show up to 19 tuples of the result relation. The heavy use of the down commands, used to display further tuples on the screen, reflects the existance of result relations with cardinality > 19.

The preference of output (formatted output to external file) against dump (unformatted output) suggests an extension to the language in terms of more report generating facilities.

The total Number of "Show mode" Commands is 9735 (15.6 % of total).

Figure 9-3: "Show" Mode Commands

comm. name	times used	%	% total	comment
up	1946	20.0	3.1	move cur. 1 line up, if any
down	6281	64.5	10.1	move cur. 1 line down, if any
left	239	2.5	0.4	move cur. 1 col. left, if any
right	704	7.2	1.1	move cur. 1 col. right, if any
store	6	0.0	0.0	this command is recognised but has presently no effect.
dump	36	0.4	0.1	unformatted output of result table to file
output	523	5.4	0.8	formatted output of result table to file

9.4.3 Update Mode Analysis

Edquse has facilities to load a database from external files. New relations and their columns may be created at load time. The use of these facilities has not been investigated, but according to the authors experience, they provide the primary way of creating relations and columns and at the same time loading the tuples from an external file. The Update mode is normally used for small (single tuple) updates.

The total number of "Update mode" Commands is 1919 (3.1 % of total). This small number is partly due to ICL/EMAS users who did not use Edquse for updates.

Figure 9-4: "Update" mode Commands

command name	times used	%	% total	comment
up	176	9.2	0.3	move cur. 1 line up, if any
down	487	25.4	0.8	move cur. 1 line down, if any
left	126	6.6	0.2	move cur. 1 col. left, if any
right	300	15.6	0.5	move cur. 1 col. right, if any
il	450	23.4	0.7	insert new tuple
dl	67	3.5	0.1	delete tuple
ic	34	1.8	0.1	insert new column
ef	98	5.1	0.2	exchange attribute value
close	169	8.8	0.3	commit update
abandon	12	0.6	0.0	abandon update

The use of the four update operations: il, dl, ic and ef, amount to a total of 646 times (33.7%). The fact that inserting lines are more frequently used than deleting lines supports the idea that databases tend to grow.

Inserting a new column (ic) is a relatively infrequent operation partly because the student's databases tend to stabilise after the initial creation of tables and columns through the bulk load command add. Being an experimental system and therefore relatively unreliable, particularly during updates, users tend to limit the interactive use of the updating facilities. Note that it is not possible to delete a column from a table.

9.5 Analysis of Queries, Sessions and Users

This section presents general statistics and frequency distributions of queries, sessions and users. This information is used in further analysis and was recorded to study user behavior in terms of number of sessions, number of queries per sessions, etc.

Again, the presentation concentrates mainly on *casual* users, indicating explicitly otherwise.

Figure 9-5: Queries, Sessions and User Totals

	Casual	Experts
Total Number of Users :	163	2
Total Number of Sessions :	1070	602
Total Number of Queries :	3971	1048

9.5.1 Sessions per User

The frequency distribution of sessions per user shown in figure 9-7, indicates that 131 users (80.4%) use Edquse less than 10 times and that 33 (20%) use it only once. The reason may be that most students work in groups, leaving the administration of the database to one person and accessing it mainly using that person's user identification. On the other hand it may be that users are disencouraged by the language. An analysis of the errors made by these users may shed some light on this issue. Better help facilities and a more responsive user interface may also improve its usability.

Sessions per user statistics, for *casual* users, are shown in figure 9-6. The two *expert* users head 237 and 365 sessions respectively.

Figure 9-6: Sessions per User Statistics

Number of users : 163 (*casual* users only)

sessions/user				
Average :	6.5			
Std dev :	10.2			
Range :	106			
Minimum :	1	observed for 33 users,	20.2%	of users
Maximum :	107	observed for 1 user,	0.1%	of users
Typical :	1	observed for 33 users,	20.2%	of users

Figure 9-7: Distribution of Sessions per User

```
Sessions/user                                                    users
-------------------------------------------------------------------------
   1 | ************************************************...* 33
   2 | ********************************************** 22
   3 | ************************************** 17
   4 | ********************************************** 20
   5 | ************************************** 18
   6 | ********************** 11
   7 | ********** 6
   8 | * 1
   9 | ***** 3
  10 | ***** 3
  11 | ********** 5
  12 | ******* 4
  13 | *** 2
  14 | *** 2
  15 | ***** 3
  17 | ***** 3
  19 | * 1
  20 | * 1
  23 | *** 2
  24 | *** 2
  31 | * 1
  34 | * 1
  41 | * 1
 107 | * 1
```

9.5.2 Queries per User

The number of *casual* users observed issuing queries (147) is less than the total number of *casual* users (163) (se figure 9-8). This may be due to groups of students working on the same database where some of the group members unsuccessfully attempt to issue correct queries on their own. Only users producing at least one correct query are considered in this analysis. It is also notable the large number of users (10.9%) issuing just one query, reinforcing the need for better help or guiding facilities.

Figure 9-8: Queries per User Statistics

```
Number of users : 147 (casual users only)

queries/user
------------
Average :   27.0
Std dev :   29.1
Range   :  186
Minimum :    1 observed for 16 users, 10.9% of users
Maximum :  187 observed for  1 user,   0.1% of users
Typical :    1 observed for 16 users, 10.9% of users
```

9.5.3 Queries per Session

The number of queries (and updates) per session indicated (804) is less than the total (1070) because only sessions including at least one query were analysed. The rest of the sessions may have been devoted to adding new tables. Figure 9-10 shows that in 43 % of the sessions users produce less than 4 queries and in 282 sessions (35.1%) only one query was issued. For *expert* users, the percentage is 57%. The reasons for this situation are manyfold: first, many people use Edquse just as a simple report generator, that is, issuing a query whose result is transferred to an external file (the command analysis in section 9.4.2 shows that the "output" command was used 523 times). Second, it has been observed that, due to the relatively unreliability of Edquse, many users updating their database will restrict updates to a minimum i.e. to one table per session (hence one 'update' run), re-entering Edquse if another table has to be updated. Third, users updating their database through 'bulk' load (add command) issue one "projecting-only" query to confirm the addition of the new data, finishing with that the session.

Figure 9-9: Queries per Session Statistics

Number of sessions : 804 (*casual* users only)

queries/session

Average:	4.9								
Std dev:	6.4								
Range :	74								
Minimum:	1	observed	for	282	sessions,	35.1%	of	sess.	
Maximum:	75	observed	for	1	session,	0.1%	of	sess.	
Typical:	1	observed	for	282	sessions,	35.1%	of	sess.	

9.6 Cpu Time Analysis

This data was primarily recorded in preparation for future studies of query optimization strategies, and to compare the efficiency of Edquse with other query evaluating systems.

The cpu time unit is the millisecond (msec) and the resolution of query time measurement is 10 msec. Points of measurements are start of query evaluation (user types 'run') and end of evaluation (system is ready to display result relation).

Figure 9-11 shows that 1321 (33.3%) queries were evaluated within 10 msec. This corresponds to the evaluation of a query for which all the necessary data was in core (probably re-evaluation) and the query was simple, with probably one operation only. Further analysis supports this observation e.g. figure 9-16 indicates that 38.8% of the queries used one operation and figure 9-14 shows that 20.8% of the queries produced a table with cardinality 0.

The average cpu time for *expert* user queries is 168 msec. Their experience and ours is that Edquse is relatively slow, however, to study the responsiveness of Edquse, this analysis should be complemented with elapsed time measurements.

Figure 9–10: Distribution of Queries per Session

```
Queries/Session                                                    sessions
-------------------------------------------------------------------------
  1 |********************************************************** ... * 282
  2 |************************** 123
  3 |****************** 81
  4 |********** 56
  5 |********* 46
  6 |******* 34
  7 |**** 21
  8 |**** 23
  9 |** 14
 10 |*** 18
 11 |** 11
 12 |** 12
 13 |** 14
 14 |** 11
 15 |** 10
 16 |* 6
 17 |* 4
 18 |* 8
 19 |* 2
 20 |* 3
 21 |* 5
 22 |* 2
 23 |* 1
 24 |* 1
 25 |* 3
 27 |* 2
 29 |* 2
 32 |* 2
 33 |* 1
 35 |* 1
 38 |* 1
 40 |* 3
 75 |* 1
```

Figure 9–11: Query Evaluation Cpu Time Statistics

```
Number of queries : 3971 (casual users only)

cpu time/query [milliseconds]

Average:     76.8
Std dev:     67.8
Range  :  1330
Minimum:        0  observed for  1321  queries, 33.3% of q.
Maximum:  1330  observed for        1  query,     0.0% of q.
Typical:        0  observed for  1321  queries, 33.3% of q.
```

9.7 Result Relation Analysis

This analysis includes the degree (figures 9–12 and 9–13) and cardinality (figure 9–14) of the result relation. The figures suggest that Edquse is used for relatively small volumes of data. 92% of the result relations have five or less columns, 20.8% have cardinality 0 and although not shown in the figures, 76.7% of the result relations have less than 20 tuples,

that is, they all fit in the screen. The high proportion of result relations with no cardinality correspond, mainly, to ICL/EMAS exercises requirying a result table with such characteristic. The result relations of *expert* users are slightly larger: average degree is 4.6 columns and average cardinality is 33.3 tuples. According to them and our own experience, Edquse is unsuitable for relations with more than 300 tuples. Beyond that, its unreliability increases and the response time worsens rapidly. However, it must be said that Edquse was designed as an experiment in screen oriented languages, not to be used as a product.

Figure 9–12: Result Relation Degree Statistics

```
Number of queries : 3971 (casual users only)

result degree/query

Average:     2.8
Std dev:     3.4
Range   :    24
Minimum:     1 observed for 1010 queries, 25.4% of q.
Maximum:    25 observed for    1 query,   0.0% of q.
Typical:     2 observed for 1093 queries, 25.4% of q.
```

Figure 9–13: Distribution of Result Relation Degree

```
Result degree/ query                                        queries
-------------------------------------------------------------------
  1 |**********************************************...* 1010
  2 |**************************************************...* 1093
  3 |**********************************...* 750
  4 |***************************** 542
  5 |************ 261
  6 |******** 161
  7 |*** 79
  8 |* 32
  9 |* 24
 10 |* 12
 11 |* 2
 12 |* 2
 13 |* 2
 25 |* 1
```

Figure 9–14: Result Relation Cardinality

```
Number of queries : 3971 (casual users only)

result cardinality/query

Average:    21.1
Std dev:    58.2
Range   :  1291
Minimum:       0 observed for 824 queries, 20.8% of q.
Maximum:    1291 observed for   2 queries,  0.0% of q.
Typical:       0 observed for 824 queries, 20.8% of q.
```

9.8 Query Structure Analysis

Figure 9–15 shows the number of relational operations used in forming a query or in preparation for an update, the number and percentage of queries having that structure and the accumulative % of all queries.

It is interesting to note that the majority of the queries have a very simple structure: the average number of operations per query is 2.2 (2.4 of *experts*). 1794 queries (45%) are over 1 table only (projection and selection). The most complex queries are one with five joins and another with 16 selections. This suggests that Edquse is used for relatively simple applications and explains the high number of queries with small cpu evaluation time.

Figure 9–15: Query Structure

Number of Operations			Number		Accumulated
project	select	join	of Queries	%	%
1	0	0	1541	38.8	38.8
1	0	1	1106	27.9	66.7
1	1	0	115	2.9	69.6
1	1	1	278	7.0	76.6
1	1	2	23	0.6	77.2
1	2	0	119	3.0	80.2
1	2	1	463	11.7	91.9
1	2	2	82	2.1	94.0
1	2	3	22	0.6	94.6
1	3	0	10	0.3	94.9
1	3	1	67	1.7	96.6
1	3	2	13	0.3	96.9
1	3	3	3	0.1	97.0
1	4	0	8	0.2	97.2
1	4	1	97	2.4	99.6
1	4	2	12	0.3	99.9
1	5	1	2	0.0	99.9
1	5	5	1	0.0	99.9
1	6	0	1	0.0	99.9
1	6	1	5	0.1	100.0
1	7	3	1	0.0	100.0
1	12	2	1	0.0	100.0
1	16	3	1	0.0	100.0

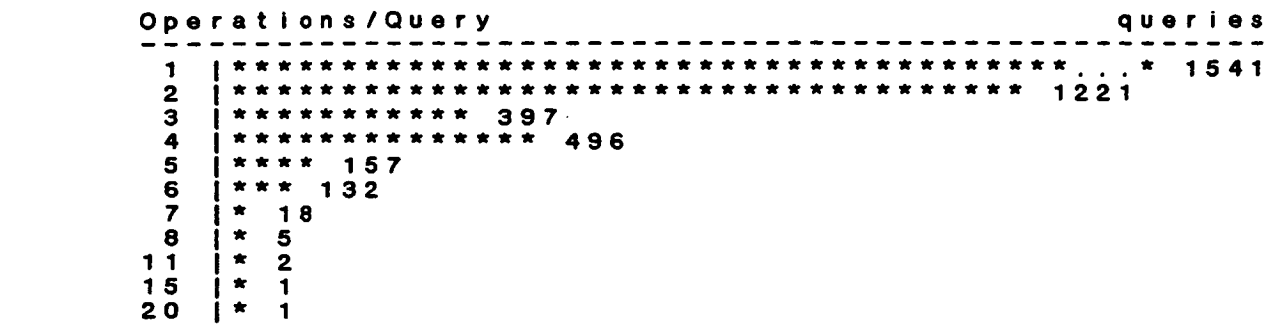
Figure 9–16: Operations Per Query Statistics

Number of queries : 3971 (*casual* users only)

operations/query

Average:	2.2						
Std dev:	2.8						
Range :	19						
Minimum:	1	observed	for	1541	queries,	38.8%	of q.
Maximum:	20	observed	for	1	query,	0.0%	of q.
Typical:	1	observed	for	1541	queries,	38.8%	of q.

Figure 9–17: Distribution of Operations per Query



9.9 Query Syntax Error Analysis

Figure 9–18 highlights the difficulty users have in defining joins. Two relations are joined by inserting the same example element in the two columns participating in the join. Since the terminal can hold up to five skeleton relations to build a query, it is cumbersome to keep track of the semantics of a query involving more than five relations. However, figure 9–15 reveals that only 1 out of 3971 queries has more than three joins, 25 have three and the rest (99.3%) have two, one or no join. Furthermore, the ICL/EMAS Edquse system indicates the domain of the columns in the skeleton tables so that users do not have to remember them, as in the VAX/VMS Edquse, in order to establish a join. However, the figures indicate that a higher proportion of EMAS users have problems with domains (14.3% of them compared to 10.9% of the total). Given that joins represent the 77.3% of the errors, we defined a different strategy for joining in our screen oriented language TABLES (see chapter 7). It would be interesting to analyse how users progress with Edquse and whether the number of errors decrease with time.

Figure 9–18: Distribution of Syntax Errors

Error type	Number of cases CASUAL	EMAS	EXPERTS
Non matching join domains	99	86	10
Tables not joined	605	410	19
No Query	119	34	4
No columns to be printed	88	70	7
Total	911	600	40

9.10 Queries vs Updates

Updates are constructed in exactly the same way as queries. The difference is that after displaying the result relation, Edquse enters update mode allowing the modification of the relation through a screen editor. In order to analyse the usage of the updating capabilities of the language and the stability of the databases under Edquse, the 'running' mode was recorded (query or update) (see also command mode analysis). The EMAS users did not update their databases which was set up by a teacher. VAX users set up and maintained their databases by themselves. From the 181 updates made by *casual* VAX users, 169 (93.4%) were effected, the rest were "abandoned" (see update mode analysis).

We indicated previously that most of Edquse updates are done via bulk loading of relations from external files. In that process, users may define a new table and its columns. Figure 9-19 and the update mode analysis suggest that the update facilities of Edquse are not much used. We also indicated some inherent Edquse problems such as its unreliability. These fact have been considered in the design of the TABLES language (see chapter7). TABLES has a more powerful screen editor and updates are applied immediately: what the user sees is what is in the database. Preliminary observations indicate that TABLES is capable of handling much larger relations than Edquse, is more reliable and faster.

Figure 9-19: Queries vs Updates

	CASUAL	EMAS	VAX
Queries:	3790 (95.4%)	2662 (100%)	1128 (86.2%)
Updates:	181 (4.6%)	-	181 (13.8%)
total :	3971	2662	1309

9.11 Raquel Analysis

We have also gathered some data on the recent use (first half of 1983) of the interactive query and update language Raquel by third year undergraduate and MSc students.

Raquel is presented in detail in chapter 7. It provides 12 operations, including project, select, join, extension, grouping, modification, ordering, outerjoin, union, difference, intersection and a tuple constructor. As with Edquse, the data is only typical of students and real life applications may probably produce different results.

9.11.1 Raquel Queries and Updates

Figure 9-20 shows the total number of queries and updates performed with Raquel. The proportion of updates is much higher than for Edquse probably because updating data and meta-data in Raquel is done using the same commands (we only recorded the interactive use of the update editor in Edquse). It may also be because Raquel simplifies and therefore

encourages the updating of the database. For example, adding a tuple to the database is done by a single line operation. Assume that the database contains a relation named x with three integer attributes: x1, x2 and x3. The Raquel operation needed to add 1 tuple where x1=1, x2=2 and x3=3 is the following:

```
C: update x := + {1,2,3}
```

With Edquse, the user has to get the x table skeleton, insert project ("p.") elements in x1, x2 and x3, run the query (syntax analysis first) and only after the result has been displayed on the screen the user can insert the line. The update must be then confirmed and applied, which is a lenghty process.

The differences in these methods and in the comparative figures have prompted a simplified update procedure for the Tables language which is similar to Edquse, as explained earlier.

From 1504 updates, 1320 (87.8%) were performed using the tuple constructor plus one set operation, as shown in the previous example and 131 (8.7%) using the modify operation. Notably, set operations were only used in updates where the tuple constructor requires them. No query included set operations. This may be due to the particular type of requests issued by students or because they are not enough familiarised with the use of set operations in query languages. Grouping functions (count, sum, min, max,etc.) were used in 14.5% of the queries (not available in Edquse), indicating their usefulness.

Figure 9-20: Raquel Queries and Updates

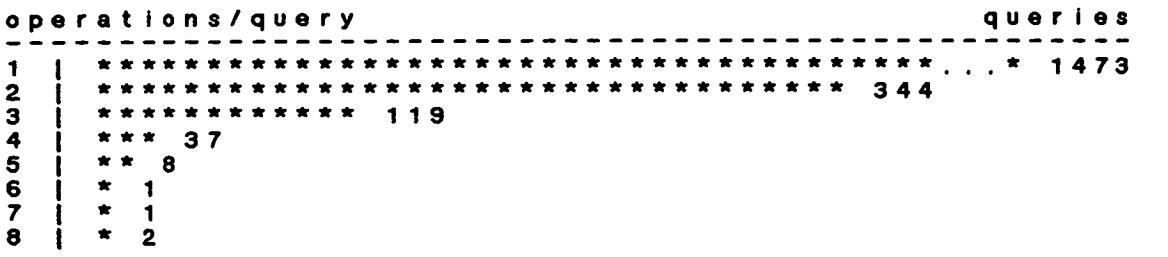
```
Total Number of Queries : 1985 (56.9%)
Total Number of Updates : 1504 (43.1%)
                          3489
```

The smaller proportion of operations/query (1.3 on average) shown in figure 9-21 with respect to Edquse (2.2 on average)) may be explained by considering that Raquel allows a query to refer to another query, hence complex queries may be built out of simple queries. This is not the case in Edquse.

Figure 9-21: Raquel: Operations per Query Statistics

```
Number of queries : 1985 (updates not included)

operations/query
Average: 1.3
Std dev: 0.6
Range  : 7
Minimum: 1 observed for 1473 queries, 74.2% of q.
Maximum: 8 observed for    2 queries    0.1% of q.
```

Figure 9-22: Raquel: Distribution of Operations per Query



9.11.2 Raquel Errors Analysis

We recorded 2312 errors which occured at all levels of Raquel: query or update formulation, use of commands, etc. The table in figure 9-23 shows the type of error and the number of times it has occurred.

Figure 9-23: Raquel: Error analysis

Error type	Occurrences
Attempt to re-use base table or view name	96
Query/update op. refers to unexisting table	267
Unexpected symbol	614
Unidentified column name	339
No arguments given for table operation	1
Unidentified inequality operator	1
Types are not comparable	66
Integer expected	7
Integer expression expected	20
Boolean expression expected	19
Tables are not union compatible(unequal degree)	1
Tables are not union compatible(unequal col type)	3
Given query name does not exist	610
Expecting a base table	50
Attempt to display a table with no columns	15
Attempt to re-define existing view	17
Attempt to update a non-base table	51
Expecting a set operation	25
Update abandoned due to integrity errors	110

The largest number of errors is due to "Unexpected symbols" (26.6%) probably because Raquel users have to type more to define a query than in Edquse and are therefore expected to make more spelling mistakes. This fact prompted some modifications in the syntax of Raquel which have yet to be tested. For example, the version used by students "entered" query mode and remained in it until an "empty" query was defined. Many users complained about this because they normally wanted to define a query and see immediately its result, before defining a new query or modifying the existing one. This caused many errors due to users wanting to display the query and Raquel expecting the definition of a new one. Furthermore, operations had to have their parameters enclosed in square brackets (e.g project [a,b,c]) which is not needed any more.

The other major cause of errors is "Given query name does not exist". This problem is mainly due to the mispelling of view names (arising from a design error in Raquel), forcing users to define views in lexicographical order if the view definition refers to other views (they are stored in B-trees and compiled at the beginning of each Raquel session, in lexicographical order). The view mechanism, unexpectedly, was heavily used.

Although not explicitly recorded, there seems to be less trouble with joins in Raquel than in Edquse. The error "Unidentified column name" may have been caused in join operations but, from our own experience, it mainly arises in project operations. Perhaps it is a characteristic of procedural queries such as Raquel that joins are more easily defined than in non procedural languages like Edquse.

We expect that the changes to the Raquel syntax prompted by observing its use, plus some desirable extensions to reduce the amount of typing by the user (e.g. such as parameterised queries) will improve its usability.

9.12 Object Reference Patterns

One of the objectives of recording query data from users is to improve the query evaluation time. For each query, the name of the columns and relations referenced in the query was recorded to investigate the frequency of object references.

Although not shown here, the results indicate that a small set of columns and relations are used more frequently in query composition than the rest. This fact can have an important effect on the performance of the system if it is properly exploited: it may automatically implement new access paths according to data usage [Dearnley 74].

We indicated in chapter 6 that an *Optimizer* software component may be included as part of the database architecture. This component will attempt to minimise the cost of processing a query. It could maintain a table indicating the name of each relation and column and the number of times it has been referenced (alone or in conjunction with another column of the same table). Once a user finishes a session (or upon explicit demand), the Optimizer may be called once again to analyse the history of references, estimate the object reference trend, and suggest (according to the the present number of indices) the creation/deletion of indices on those columns above/below a given threshold [Stocker 73]. This process can be done automatically, without consulting the user.

Many parameters may intervene in this analysis, e.g. the cardinality of the relations involved in the queries, some priorities set up by the users or the number of the already existing indices. As databases grow and more users have access to them for different purposes, this technique may help adapting the database to the users to achieve better responsiveness. However, it may have to be finely tuned to avoid potential problems such as overpopulation of indices or too frequent modification of them.

What we have sketched above corresponds to a database reorganization at the *string* level, which is one of the many possible reorganizations discussed in [Sockut 79].

9.13 Conclusions

The recording of the usage of a database language can help designers to improve not only the language but the whole database system. This may be a dynamic process in which the system adapts to its users and makes itself more efficient as a tool.

Although Edquse was designed for experiments only, the analysis has suggested many areas for improvements that would upgrade it into a useful tool. The principal problems seem to be its lack of help facilities, unreliability, and the limited number of objects it can handle. The screen oriented approach for query formulation and updates provides ample room for improvements which have influenced the design of the language TABLES (see chapter 7).

A framework for recording and analysing user queries under Edquse has been set up. Data is gathered from its use and stored in a central file which is processed by the query languages and report generator described in chapter 7. Most of the data presented in this chapter is typical only of students. Other uses in a wide range of real situations, as well as comparison with other languages are needed to be conclusive about the many suggestions we have made.

The preliminary analysis of the data has provided useful feedback toward a better understanding of the Edquse and Raquel users and the performance of the system. Many enhancements of the whole system were suggested from the analysis. This study has also incentivated further data recording and more elaborate analysis.

Chapter 10
Conclusions

10.1 Research Aims and Proposals

The purpose of this research was to investigate the feasibility of applying database technology, at a moderate cost, in small but potentially expanding applications. The evolution of an application may relate to larger volumes of data, to increasingly complex processing requirements or to the diversification of activities. This evolution may be reflected in the systematic building of a database environment.

The adequate administration and use of information are growing rapidly in importance in many activities with even small volumes of data. However, database technology has been traditionally oriented toward large applications. We have therefore investigated the needs of small applications which would certainly benefit from more flexible and economic data processing tools.

We indicated in the introductory chapters some trends that are rapidly broadening the possibilities for database technology to become a useful tool for a wide range of applications and users. Hardware advances and dropping prices, and the increasing interest in providing software products that are appealing to non-experts imposed two requirements on our research.

One requirement was physical independence to cope with hardware advances. The other was the provision of an internal flexibility that would make it easy to support a graded set of compromises between end user functionality and the complexity of the interface.

A variety of existing database systems were analysed. It was shown that the more complete of them offer, in general, a ready-made choice of user interfaces aimed at large databases to which the users have to adapt. Most of them are tied to a particular make of hardware, are too resource demanding or over-complex for small applications. On the other hand, the smaller systems are too restricted, limiting the potential growth of an application. Finally, most of the systems do not provide adequate facilities for re-using parts of their software, modifying or experimenting with it, as a database environment would require.

We set out in chapter 3 to investigate the engineering requirements for building database

software. We concluded that an adequate integration of well tried techniques such as functional decomposition, re-usability of tools, rapid prototyping and the identification of internal software standards could provide the basis for our research framework. By applying these techniques, the many advantages in developing the re-usable experience encapsulated in the software would have a positive impact in building economic and more reliable database software. The challenge was to identify a near optimum decomposition of a DBMS into components. At present, the reported experiences suggest that building a DBMS demands a massive software effort.

The need to support a wide range of approaches at both the user and hardware level led us to investigate existing data models. (chapter 4). It was found that a modified subset of the extended relational model RM/T [Codd 79] offers adequate data independence, flexibility and power and it was chosen as the internal data model. On top of it, both functional and relational models may coexist. This was shown for a subset of the functional languages EFDM [Kulkarni 83] and FQL [Buneman 82].

The internal data model is represented by three internal standards: a structural standard consisting of an Internal Conceptual Schema (ICS) supporting a variety of external conceptual schemas based on possibly different data models and two operational standards related to the ICS. One is a data manipulation language that operates on objects defined in the internal conceptual schema. The other is a common Internal Query Language to which user interfaces map their queries which are processed by a single common query evaluator component. These internal standards and a number of cooperating software components centered around them were defined. They constitute the proposed database architecture.

Each software component embodies a number of functions related to a specific database task. A "minimum" functionality for each of the components was defined that can support a prototype system which can be systematically extended and modified according to the specific needs and preferences of the users and the requirements of the application.

The demonstration of the feasibility of the proposals was achieved by implementing a database environment comprising a variety of end user interfaces. It consists of three relational interfaces, one functional interface and a report generator. The query systems are RAQUEL and TABLES and a query monitor used while building the system. The functional interface is FQL (see chapter 7).

Our experience indicates that after the basic software framework is accomplished, new user interfaces can be provided in working condition in approximately one man/week. This is a demonstration of the effectiveness of the software architecture chosen.

Software economy was achieved mainly by a combination of two factors. First by the re-use of software components and the cumulative experience gained in that process. Second, by working with adequate tools, particularly a programming language such as PS-algol which relieves the practitioner from some of the hardest tasks in implementing database software that is, the uniform management of transient and persistent data. This thesis also constitutes one of the earliest evaluations of a persistent language (chapter 5). Having used it for a substantial project, we believe it is a significant language development.

It was shown that the usability of user interfaces can be improved by a step–wise adaptation to the user's needs and preferences. This is done by first providing an interface according to the designer's intuition of what is convenient for the users. This intuition is then continuously refined by observing, recording and analysing how query languages are used (as well as by performing controlled experiments). The data recorded should include errors, query and update structures, result relation structures, performance evaluation parameters, etc. The analysis of the data may help in at least two ways: it may improve the user interface and it may improve the efficiency of the system by applying data restructuring and optimization techniques. By recording the use of a system, it can automatically modify itself as result of analysing its recent usage history. How to integrate these facilitites and how to automate these tasks are both worth investigating.

10.2 Directions for Further Research

All of the topics addressed in this thesis offer ample room for more research and experimentation. Our proposals derived from both observing the work of other practitioners and from our own experiments. More cross fertilization from the many areas involved in the process of making tools for humans is required. This is necessary to improve the quality, scope and economy in our approach. We pinpoint a few areas which clearly need more contributions:

A more comprehensive analysis of the EFDM structures and mappings into the RM/T ICS is required. In particular views and updates deserve a more thorough investigation. An implementation of the methods presented should help in clarifying this process. Although the relational and functional based query languages are claimed to be easy to use and have been successfully used by non–experts, more research is needed to improve the present user interfaces based on those models. Many suggestions in this direction were made in chapter 7. The present implementation would provide a good foundation for this research.

One of the potential research problems mentioned in chapter 3 was that attempting to incorporate models other than the relational could prove a limitation with respect to the freedom for experimenting with different alternatives. This problem arises from the additional complexity that has to be handled by the components. This limitation effectively hindered some of the efforts, particularly in the attempts to incorporate query optimization strategies. It also affected negatively the prospects of extending the internal Query Language into a more complete body of instructions. Building an optimizer component would be a high priority in developing the system.

The software components of the architecture are by definition, candidates for change. It will be desirable to extend the existing ones, formalise and refine their interfaces and build and accomodate new ones. We mentioned the need for View, Protection and Security components as well as a Query Optimizer. The common Query Evaluator ought to provide alternative evaluation strategies according to the query requirements and data access paths. The present Logical Storage Handler should be provided with structures offering more compactness and faster access. These structures could be dynamically modified according to statistic analysis of the use of the database objects. The present system lacks concurrency but it is expected that the underlying system (PS–algol) will provide it.

The design and construction of Operating Systems has benefited greatly from a study of how they should be decomposed and from a study of the function, method of implemention and performance of the identified components. It is believed that a similar study is needed in database systems design and construction. Some evidence of such study is beginning to appear in the literature [Bernstein 81, Jarke 82] and [Sevcik 83]. This thesis is a contribution to that study, which we hope will be continued both to improve the quality and economy of DBMS.

Considering the variety of research being pursued in information sciences and the beneficial cross fertilization occuring in the many areas involved, it is expected that database technology will continue developing at a fast rate. Since progress derives from the quality of the information available from our collective experiences, the beneficiaries of information science research can only be all humankind.

References

[Anderson 80] Anderson, B.
Programming in the Home of the Future.
International Journal Man-Machine Studies 12:341-365, 1980.

[Astrahan 79] Astrahan, M.M. et.al.
System R: A Relational Database Management System.
IEEE Computer 12(5):42-48, May, 1979.

[Atkinson 78a] Atkinson, M.P.
Programming Languages and Databases.
In *4th International Conference on Very Large Databases, West-Berlin, Germany*, pages 408-419. IEEE, September, 1978.

[Atkinson 78b] Atkinson, M.P. Martin, J. and Jordan, M.
A Uniform Modular Structure for Databases and Programs.
Technical Report, University of Edinburgh. Computer Science Department, October, 1978.
Internal Report CSR-33-78.

[Atkinson 79] Atkinson, M.P.
Database Systems.
Technical Report, University of Edinburgh. Computer Science Department, October, 1979.
Internal Report CSR-43-79.

[Atkinson 81a] Atkinson, M.P (editor).
Data Bases.
In *Infotech State of the Art Report. Database*, . Pergamon Infotech, 1981.
Series 9, Number 8.

[Atkinson 81b] Atkinson, M.P. Chisholm, K.J. and Cockshott, W.P.
Nepal - the New Edinburgh Persistent Algorithmic Language.
In *Infotech State of the Art Report. Database*, pages 299-318. Pergamon Infotech, 1981.
Series 9, Number 8.

[Atkinson 82] Atkinson, M.P. Gray, P.D.M and Hepp, P.E.
Proteus Working Paper E2.
Technical Report, University of Edinburgh, Computer Science Department, 1982.

[Atkinson 83a] Atkinson, M.P. Chisholm, K.J. and Cockshott, W.P.
CMS-A Chunk Management System.
Software-Practice and Experience :273-285, 1983.

[Atkinson 83b] Atkinson, M.P. Chisholm, K.J. and Cockshott, W.P.
Algorithms for a Persistent Heap.
Software-Practice and Experience :259-271, 1983.

[Atkinson 83c] Atkinson, M.P et. al.
An Approach to Persistent Programming.
1983.
To be published in The Computer Journal.

[Backus 78] Backus, J.
Can Programming be Liberated from the Von Neumann Style?
ACM Communications 21(8):613-641, August, 1978.

[BCS 81] Samet, P.A. (editor).
Query Languages: A Unified Approach.
Heyden and Son Ltd., 1981.
Monographs in Informatics.

[Belady 81] Belady, L.A.
Manual, Machine-Aided and Mechanical Processes in Software Development and Maintenance.
In *Infotech State of the Art Report. System Design*, pages 177-183. Pergamon Infotech, 1981.
Number 6, Series 9.

[Bell 80] Bell, R.M.A.
Automatic Generation of Programs for Retrieving Information from Codasyl Data Bases.
PhD thesis, University of Aberdeen, 1980.

[Benbasat 81] Benbasat, I. Dexter, A.S. and Masulis, P.S.
An Experimental Study of the Human/Computer Interface.
ACM Communications 24(11):752-762, November, 1981.

[Bernstein 81] Bernstein, P. and Goodman, N.
Concurrency Control in Distributed Database Systems.
ACM Computing Surveys 13(2):185-221, June, 1981.

[Blasgen 76] Blasgen, M.W. and Eswaran, K.P.
A Comparison Of Four Methods For The Evaluation Of Queries In A Relational Data Base System.
Technical Report, IBM Research Laboratory, San Jose, California, February, 1976.
RJ 1726 (25344).

[Blundell 83] Blundell, G.S.
Personal Computers in the Eighties.
Byte 8(1):166-182, January, 1983.

[Boehm 79] Boehm, B.W.
Software Engineering - As It Is.
4th International Conference on Software Engineering Munich, Germany :11-21, September, 1979.

[Bontempo 80] Bontempo, C.J.
Feature Analysis of Query-by-Example.
In Schmidt, J.W. and Brodie, M.L. (editors), *Relational Database Systems*, pages 409-437. Springer-Verlag, 1980.

[Borkin 78] Borkin, S.A.
Data Model Equivalence.
In *4th International Conference on Very Large Data Bases, West-Berlin, Germany*, pages 526-534. IEEE, September, 1978.

[Brodie 80] Brodie, M.L.
Standardization and the Relational Approach to Databases: an ANSI Task Group Status Report.
In *6th International Conference on Very Large Date Bases, Montreal, Canada*, pages 326-327. IEEE, October, 1980.

[Brooks 79] Brooks, F.P.
The Mythical Man-Month.
Addison-Wesley, 1979.

[Buneman 79] Buneman, P. and Frankel, R.E.
FQL — A Functional Query Language.
In *Proceedings of the International Conference on the Management of Data*, pages 52–58. 1979.

[Buneman 82] Buneman, P. Frankel, R.E. and Nikhil, R.
An Implementation Technique for Database Query Languages.
ACM Transactions on Database Systems 7(2):164–186, June, 1982.

[Buneman 83] Buneman, P. and Davidson, S.B.
Databases and Programming Languages for Heterogeneous Database Management Systems.
Technical Report, University of Pennsylvania. Department of Computer and Information Science, 1983.

[Chan 81] Chan, A. et.al.
ADAPLEX DBMS Design Specification.
Technical Report, Computer Corporation of America, June, 1981.

[Chang 81] Chang, N.S. and Fu, K.S.
Picture Query Languages for Pictorial Data-Base Systems.
IEEE Computer 14(11):23–33, November, 1981.

[Codd 70] Codd, E.F.
A Relational Model for Large Shared Databanks.
ACM Communications 13(6):377–387, June, 1970.

[Codd 71] Codd, E.F.
Relational Completeness of Data Base Sublanguages.
In Randall Rustin (editor), *Courant Computer Science Symposium 6. Data Base Systems*, pages 65–98. Prentice Hall, 1971.

[Codd 79] Codd, E.F.
Extending the Database Relational Model to Capture More Meaning.
ACM Transactions on Database Systems 4(4):397–434, December, 1979.

[Codd 80] Codd, E.F.
Data Models in Database Management.
In *Proceedings of the Workshop in Data Modelling*, pages 112–114. IFIP North-Holland, 1980.

[Codd 82] Codd, E.F.
Relational Database: A Practical Foundation for Productivity.
ACM Communications 25(2):109–117, February, 1982.

[Cole 82] Cole, A.J. and Morrison, R.
An Introduction To Programming With S-algol.
Cambridge University Press, 1982.

[Comer 79] Comer, D.
The Ubiquitous B-Tree.
ACM Computing Surveys 11(2):121–137, June, 1979.

[Cuff 79] Cuff, R.
Database Query Systems for the Casual User.
Master's thesis, Dept. of Electrical Engineering Science, University of Essex, March, 1979.

[Dahl 72] Dahl, O.J. Dijkstra, E.W. and Hoare, C.A.R.
Structured Programming.
Academic Press, 1972.
A.P.I.C. Studies in Data Processing No.8.

[Date 81] Date, C.J.
The Systems Programming Series: An Introduction to Database Systems.
Addison-Wesley, 1981.
Third Edition.

[Date 83] Date, C.J.
The Systems Programming Series. Volume 2: *An Introduction to Database Systems.*
Addison-Wesley, 1983.

[Dearnley 74] Dearnley, P.A.
A Model of a Self-organising Data Management System.
The Computer Journal 17(1):13-16, 1974.

[Dearnley 83] Dearnley, P.A. and Mayhew, P.J.
In Favour of System Prototypes and their Integration into the Systems Development Cycle.
The Computer Journal 26(1):36-42, February, 1983.

[Demers 81] Demers, R.A.
System Design for Usability.
ACM Communications 24(8):494-501, August, 1981.

[Dieckmann 81] Dieckmann, E.M.
Three Relational DBMS.
Datamation 27(10):137-148, September, 1981.

[Dijkstra 72] Dahl, O.-J. Dijkstra, E.W and Hoare, C.A.R.
Structured Programming.
Academic Press, 1972.

[Driver 80] Driver, B.H.
Feature Analysis of ORACLE.
In Schmidt, J.W. and Brodie, L.M. (editors), *Relational Database Systems,* pages 288-331. Springer-Verlag, 1980.

[Eswaran 75] Eswaran, K.P. and Chamberlin, D.D.
Functional Specifications of a Subsystem for Data Base Integrity.
In *1st International Conference on Very Large Data Bases.* September, 1975.

[Gladden 82] Gladden, G.R.
Stop the Life Cycle, I Want to Get Off.
ACM Software Engineering Notes 7(2):35-39, April, 1982.

[Gomaa 81] Gomaa, H. and Scott, D.B.H.
Prototyping as a Tool in the Specification of User Requirements.
In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California,* pages 333-339. Computer Society Press, 1981.

[Gradwell 81] Gradwell, D.J.L.
The Role of a Data Dictionary System in Supporting a Database.
In *Infotech State of the Art Report. Database,* pages 411-424. Pergamon Infotech, 1981.
Series 9, Number 8.

[Gray 82] Gray, P.M.D.
The Group-By operation in Relational Algebra.
Technical Report, University of Aberdeen. Dept. of Computer Science, 1982.

[Hall 75] Hall, P.A.V.
Optimisation of a Single Relational Expression in a Relational Data Base System.
Technical Report, IBM UK Scientific Centre, June, 1975.
UKSC 0076.

[Hall 76] Hall, P.V.
Relations and Entities.
In *Modelling in Data Base Management System,* pages 210-220. IFIP North-Holland, 1976.

[Hammer 75] Hammer, M.M. and McLeod, D.J.
Semantic Integrity in a Relational Data Base System.
In *1st International Conference on Very Large Data Bases.* September, 1975.

[Henderson 81] Henderson, P. (Editor).
Requirements Analysis and Specifications.
In *Infotech State Of The Art Report. System Design*, . Pergamon Infotech, 1981.
Series 9, Number 6.

[Hepp 83] Hepp, P.E.
RAQUEL, TABLES and Report Generator User Manuals.
Technical Report, University of Edinburgh, 1983.
Persistent Programming Notes, Computer Science Department.

[Hoare 76] Hoare, C.A.R.
The Engineering of Software: A Startling Contradiction.
Computers and People , July, 1976.

[Jacobs 83] Jacobs, B.E. and Walczac, C.A.
A Generalized Query-by-Example Data Manipulation Language Based on Database Logic.
IEEE Transactions on Software Engineering SE-9(9):40-57, January, 1983.

[Jarke 82] Jarke, M. and Koch, J.
A Survey of Query Optimization in Centralized Database Systems.
Technical Report, Center for Research on Information Systems, New York University, November, 1982.
CRIS 44, GBA 82-73 (CR).

[Kent 76] Kent, W.
New Criteria for the Conceptual Schema.
In *Systems for Large Data Bases*, pages 1-12. IFIP North-Holland, 1976.

[Kerschberg 76] Kerschberg, L., Klug, A. and Tsichritzis, D.
A Taxonomy of Data Models.
In *Systems for Large Data Bases*, pages 43-64. IFIP North-Holland, 1976.

[Kersten 81] Kersten, M.L. and Wasserman, A.I.
The Architecture of the Plain Data Base Handler.
Software-Practice and Experience 11:175-186, 1981.

[Kim 79] Kim, W.
Relational Database Systems.
ACM Computing Surveys 11(3):185-210, September, 1979.

[King 81] King, T.J. and Moody, J.K.M.
The Design and Implementation of CODD.
Technical Report, Computer Laboratory, Cambridge, England, 1981.

[Kinsley 80] Kinsley, K.C.
Feature Analysis of SYSTEM R.
In Schmidt, J.W. and Brodie, L.M. (editors), *Relational Database Systems*, pages 519-559. Springer-Verlag, 1980.

[Krakowiak 81] Krakowiak, S.
Structural Issues in the Design of Large Systems.
In *Infotech State Of The Art Report. System Design*, pages 245-261. Pergamon Infotech, 1981.
Series 9, Number 6.

[Krass 81] Krass, P. and Wiener, H.
The DBMS Market is Booming.
Datamation 27(10):153-170, September, 1981.

[Kulkarni 83] Kulkarni, K.G.
Extended Functional Data Model - User Manual.
Technical Report, University Of Edinburgh. Computer Science Department, 1983.

[Lacroix 81] Lacroix, M.
Feature Analysis of RAPPORT.
In Schmidt, J.W. and Brodie, L.M. (editors), *Relational Database Systems*, pages 477-518. Springer-Verlag, 1981.

[Logiquest 82] Software Products International.
LogiQuest III Relational Database Management System User's Manual.
Technical Report, Software Products International, Inc. (SPI), 1982.

[McCracken 82] McCracken, D.D. and Jackson, M.A.
Life-Cycle Concept Considered Harmful.
ACM Software Engineering Notes 7(2):29-32, April, 1982.

[McDonald 81] McDonald, N. and McNally, J.
Feature Analysis of Ingres.
In Schmidt, J.W. and Brodie, L.M. (editors), *Relational Database Systems*, pages 158-181. Springer-Verlag, 1981.

[McDuff 80] McDuff, A.D.
Implementation of Query-By-Example.
Technical Report, University of Edinburgh, Computer Science Department, May, 1980.
CS4 Project Report.

[McLeod 75] McLeod, D. and Meldman, M.
RISS: A Generalized Minicomputer Database Management System.
In *National Computer Conference*, pages 397-402. AFIPS, 1975.

[Moran 81] Moran, T.P.
Guest Editor's Introduction: An Applied Psychology of the User.
ACM Computing Surveys 13(1):1-11, March, 1981.

[Morrison 81] Morrison, R.
S-Algol: Reference Manual
Department of Computer Science, University of St Andrews, 1981.
Internal Report CSR-80-81.

[Morrissey 79] Morrissey, J.H. and Wu, L.S.-Y.
Software Engineering...An Economic Perspective.
In *Proceedings 4th International Conference on Software Engineering, Munich, Germany*, pages 412-422. IEEE, 1979.

[Musgrave 81] Musgrave, B.
A Sequel for Data Bases.
Datamation 27(3):69-70, March, 1981.

[NBS 81] Brandstad, M.A. and Richards, W. (editors).
NBS Workshop Report on Programming Environments.
ACM Software Engineering Notes 6(4):1-51, August, 1981.

[Nijssen 76] Nijssen, G.M.
A Gross Architecture for the Next Generation Database Management Systems.
In *Modelling in Data Base Management Systems*, pages 1-24. IFIP North-Holland, 1976.

[Nikhil 82] Nikhil, R.
RDB - A Relational Database Management System.
Technical Report, Department of Computer Science and Information Science, University oh Pennsylvania, January, 1982.
User Manual.

[Parnas 72] Parnas, D.L.
A Technique for Software Module Specification With Examples.
ACM Communications 15(5):330-336, May, 1972.

[Pirotte 78] Pirotte, A.
Linguistics Aspects of High-Level Languages.
In *Infotech State of The Art Report. Data Base Technology*, pages 271-300. Infotech International, 1978.
Volume 2, Invited Papers.

[Reisner 75] Reisner, P.
Human Factors Evaluation of two Database Query Languages- Square and Sequel.
In *National Computer Conference*, pages 447-452. AFIPS, 1975.

[Reisner 81] Reisner, P.
Human Factor Studies of Database Query Languages: A Survey and Assesment.
ACM Computing Surveys 13(1):13-31, March, 1981.

[Robertson 80] Robertson, P.S.
The IMP-77 Language.
Technical Report, University of Edinburgh, November, 1980.
Internal Report CSR-19-77 Third Edition.

[Rowe 81] Rowe, L.A. and Stonebraker, M.
The Architecture of Future Database Systems.
ACM SIGMOD Record 11(1):30-44, January, 1981.

[Schmid 75] Schmid, H.A. and Swenson, J.R.
On the Semantics of the Relational Model.
In *Proceedings of the International Conference on the Mangement of Data*, pages 211-223. ACM SIGMOD, 1975.

[Schmidt 77] Schmidt, J.W.
Some High Level Language Constructs for Data of Type Relation.
ACM Transactions on Database Systems 2(2):247-261, September, 1977.

[Schmidt 83] Schmidt, J.W. and Brodie, L.M. (editors).
Relational Database Systems.
Springer-Verlag, 1983.

[Senko 77] Senko, M.E.
FORAL- LP: Making Pointed Queries with a Light Pen.
In Gilchrist, B. (editor), *IFIP*, pages 635-640. North-Holland, 1977.

[Serlin 83] Serlin, O.
Selecting Super Chips and Super Micros.
Datamation , February, 1983.

[Sevcik 83] Sevcik, K.C.
Comparison of Concurrency Control Methods Using Analytic Models.
To be published in IFIP 1983.

[Shipman 81] Shipman, D.W.
The Functional Data Model and The Data Language Daplex.
ACM Transactions on Database Systems 6(1):140-173, March, 1981.

[Shneiderman 79] Shneiderman, B.
Human Factors Experiments in Designing Interactive Systems.
IEEE Computer :9-19, December, 1979.

[Smith 77] Smith, J.M. and Smith, D.C.P.
Database Abstractions: Aggregation and Generalization.
ACM Transactions on Database Systems 2(2):105-133, June, 1977.

[Smith 80] Smith, J.M. et.al.
Multibase--Integrating Heterogeneous Distributed Data Bases.
Technical Report, Computer Corporation of America, November, 1980.

[Sockut 79] Sockut, G.H. and Goldberg, R.P.
Database Reorganization-Principles and Practice.
ACM Computing Surveys 11(4):371-395, December, 1979.

[Stocker 73] Stocker, P. and Dearnley, P.A.
Self Organising Data Management Systems.
The Computer Journal 16(2):100-105, 1973.

[Stocker 81] Stocker, P.
Canonical Schemata, Canonical Queries and Standardisation.
In *Infotech State Of The Art Report. Database*, pages 459-467. Pergamon Infotech, 1981.
Series 9, Number 8.

Stocker 83] Stocker, P. et. at.
Proteus: A Distributed Database System.
1983.
To be published in Database - Role and Structure, CUP.

Stonebraker 76]
Stonebraker, M. Wong, E. Kreps, P. and Held, G.
The Design and Implementation of INGRES.
ACM Transactions on Database Systems 1(3):189-222, September, 1976.

[Stonebraker 80]
Stonebraker, M.
Retrospection on a Database System.
ACM Transactions on Database Systems 5(2):225-240, June, 1980.

[Thomas 75] Thomas, J.C and Gould, J.D.
A Psychological Study of Query By Example.
In *National Computer Conference*, pages 439-445. AFIPS, 1975.

[Todd 76] Todd, S.J.P.
The Peterlee Relational Test Vehicle - A System Overview.
IBM Systems Journal 15(4):285-308, 1976.

[Tsichritzis 78] Tsichritzis, D.C. and Klug, A. (editors).
The ANSI/X3/SPARC DBMS Framework Report of the Study Group On Database Management Systems.
Information Systems 3:173-191, 1978.

[Tsichritzis 82] Tsichritzis, D.C. and Lochovsky, F.H.
Data Models.
Prentice-Hall, 1982.

[Ullman 82] Ullman, J.D.
Principles of Database Systems.
Pitman, 1982.
Second Edition.

[Vandijck 78] Vandijck, E.
An Overview of Current Relational Data Base Query Languages.
In *Infotech State Of The Art Report. Data Base Technology*, pages 423-441.
Infotech International, 1978.
Volume 2, Invited Papers.

[Wasserman 81] Wasserman, A.I.
Towards Integrated Software Development Environments.
In *Infotech State Of The Art Report. System Design*, pages 379-403.
Pergamon Infotech, 1981.
Series 9, Number 6.

[Wasserman 82] Wasserman, A.I.
The Future of Programming.
ACM Communications 25(3):196-206, March, 1982.

[Wegner 82] Wegner, P.
Toward Capital-Intensive Engineering of Information.
ACM Software Engineering Notes 7(4):24-33, October, 1982.

[Weinberg 71] Weinberg, G.M.
The Psychology of Computer Programming.
Van Nostrand Reinhold, 1971.

[Welty 81] Welty, C. and Stemple, D.
Human Factors Comparison of a Procedural and a Nonprocedural Query Language.
ACM Transactions on Database Systems 6(4):626-649, December, 1981.

[Wiederhold 77] Wiederhold, G.
Database Design.
McGraw-Hill Kogakusha, 1977.

[Wilson 81] Wilson, A.G.
EDQUSE, Edinburgh Database Query Language with Update by Screen Edit.
Technical Report, University of Edinburgh, Computer Science Department, May, 1981.
CS4 Project Report.

[Yao 79] Yao, S.B.
Optimization of Query Evaluation Algorithms.
ACM Transactions on Database Systems 4(2):133-157, June, 1979.

[Zloof 75] Zloof, M.M.
Query By Example.
In *National Computer Conference*, pages 431-438. AFIPS, 1975.

[Zloof 77] Zloof, M.M.
Query By Example: A Database Language.
IBM Systems Journal 16(4):324-343, 1977.

[Zloof 81] Zloof, M.M.
QBE/OBE: A Language for Office and Business Automation.
IEEE Computer 14(5):13-22, May, 1981.