

# The design and implementation of a relational database management system

**Richard Leyton**

For further information please visit

<http://leap.sourceforge.net>

*This document is copyright ©[Richard Leyton](#), 1995, all rights reserved.*

*You may freely distribute this document for non-commercial use. No charge should be made other than to cover reproduction costs. Any use of this document other than for legitimate educational purposes must be discussed with the author first.*

**Part 1**

*For my parents,  
for everything.*

# Acknowledgements

I wish to express my gratitude to Dr Stefan Stanczyk for his much valued help throughout the project. Realistic from the outset.

A great debt of thanks is owed to friends who have put up with my mutterings, clutter and surprise requests throughout the duration of the project.

# Abstract

The intent is to design and implement a sizeable subset of the criterion of a relational database management system:

- Structural aspects of the relational model.
- A data sublanguage at least as powerful as the relational algebra.
- A framework for further enhancement.

Drawing upon the examples provided in academic and commercial relational database management systems, a system referred to as LEAP, is proposed, implemented and tested. LEAP offers the user an opportunity to explore and experiment with the relational algebra using a specified language.

The deliberately modular design enables enhancements and modifications to be added at a later date, and examples of such are given.<sup>4</sup>

# Table of Contents

<b>1. INTRODUCTION.....</b>	<b>1</b>
1.1 BASIS AND AIMS FOR THE PROJECT .....	1
1.2 DOCUMENT STRUCTURE.....	1
<b>2. THE SCOPE OF THE SYSTEM.....</b>	<b>3</b>
2.1 DEFINITION OF A RELATIONAL DATABASE MANAGEMENT SYSTEM .....	3
2.1.1 <i>The structural aspects of the relational model</i> .....	3
2.1.2 <i>The insert-update-delete rules</i> .....	4
2.1.3 <i>A data sub-language</i> .....	4
2.2 PROJECT SCOPE.....	5
<b>3. SYSTEM DESIGN.....</b>	<b>6</b>
3.1 INTRODUCTION .....	6
3.2 THE DESIGN OF THE STRUCTURAL ASPECTS OF THE RELATIONAL MODEL .....	6
3.2.1 <i>Introduction</i> .....	6
3.2.2 <i>The database structure</i> .....	7
3.2.3 <i>The relation structure</i> .....	9
3.2.4 <i>The heading structure</i> .....	11
3.2.5 <i>The body structure</i> .....	13
3.2.6 <i>Summary</i> .....	16
3.3 THE DATA SUB-LANGUAGE .....	17
3.3.1 <i>Introduction</i> .....	17
3.3.2 <i>General requirements</i> .....	20
3.3.3 <i>Project</i> .....	20
3.3.4 <i>Union</i> .....	21
3.3.5 <i>Intersect</i> .....	22
3.3.6 <i>Difference</i> .....	22
3.3.7 <i>Product</i> .....	23
3.3.8 <i>Join</i> .....	24
3.3.9 <i>Select</i> .....	25
3.4 THE INTERFACE .....	25
3.4.1 <i>Introduction</i> .....	25
3.4.2 <i>Language</i> .....	26
3.4.3 <i>Syntax</i> .....	27
3.4.4 <i>Additional Functionality</i> .....	27
3.4.5 <i>Approach</i> .....	28
3.4.6 <i>Syntax transformation</i> .....	29
3.4.7 <i>User support</i> .....	30
3.4.8 <i>Conclusions</i> .....	31
3.5 THE ENHANCEMENT FRAMEWORK .....	31
3.5.1 <i>Introduction</i> .....	31
3.5.2 <i>Structure</i> .....	31
3.5.3 <i>Efforts</i> .....	32
3.5.4 <i>Proposed examples</i> .....	32
3.5.5 <i>Use of Methods</i> .....	33
3.5.6 <i>Conclusions</i> .....	34
3.6 DESIGN SUMMARY .....	34
<b>4. IMPLEMENTATION ISSUES.....</b>	<b>35</b>
4.1 INTRODUCTION .....	35
4.2 THE IMPLEMENTATION ENVIRONMENT.....	35
4.2.1 <i>Introduction</i> .....	35
4.2.2 <i>Available resources</i> .....	35
4.2.3 <i>Knowledge of resources</i> .....	35
4.2.4 <i>Prior experience</i> .....	36

4.2.5 Personal preference.....	36
4.2.6 Conclusion.....	36
4.3 IMPLEMENTATION LANGUAGE.....	36
4.3.1 Introduction.....	36
4.3.2 Available resources.....	37
4.3.3 Knowledge of the languages.....	37
4.3.4 Conclusion.....	37
4.4 IMPLEMENTATION APPROACH.....	37
4.4.1 Introduction.....	37
4.5 IMPLEMENTATION ISSUES.....	38
4.5.1 Introduction.....	38
4.5.2 Relation naming.....	38
4.5.3 Command processing.....	39
4.5.4 Data structures.....	42
4.5.5 File storage.....	52
4.5.6 Operational structure.....	55
4.5.7 User support.....	57
4.6 CONCLUSION.....	59
<b>5. TESTING.....</b>	<b>60</b>
5.1 INTRODUCTION.....	60
5.2 TEST DATABASE.....	61
5.2.1 Book.....	61
5.2.2 Subject.....	61
5.2.3 Index.....	62
5.2.4 Auction.....	62
5.2.5 A.....	62
5.2.6 B.....	62
5.3 INDIVIDUAL OPERATORS.....	62
5.3.1 Print.....	62
5.3.2 Project.....	63
5.3.3 Union.....	63
5.3.4 Intersection.....	63
5.3.5 Difference.....	63
5.3.6 Product.....	63
5.3.7 Divide.....	63
5.3.8 Select/Restrict.....	64
5.3.9 Join.....	64
5.4 COMBINED OPERATIONS.....	64
5.4.1 Introduction.....	64
5.4.2 Project/Divide/Project.....	64
5.4.3 Select/Join/Project.....	65
5.4.4 Select/Project.....	65
5.4.5 Restrict/Project/Join/Project/Join.....	65
5.4.6 Select/Join/Project/Join/Select/Project.....	65
5.4.7 Nested Project/Select.....	66
5.4.8 Nested Project/Join/Project/Select.....	66
5.5 ADDITIONAL OPERATIONS.....	66
5.5.1 Indexing.....	66
5.6 STRESS.....	67
5.6.1 Maximum degree.....	67
5.6.2 Maximum cardinality.....	67
5.6.3 Load.....	67
5.6.4 Error handling.....	67
5.7 MISCELLANEOUS OPERATIONS.....	68
5.7.1 Scripts.....	68
5.7.2 Help facility.....	68
5.7.3 User creation of relations.....	68
5.7.4 User addition of tuples.....	68

5.7.5 Displaying structures.....	68
5.7.6 Redirected output file .....	68
5.7.7 Report file.....	68
5.7.8 Report file, with a nested expression.....	69
5.8 INTERNET RELEASE.....	69
5.9 CONCLUSION .....	69
<b>6. CONCLUSIONS .....</b>	<b>70</b>
6.1 INTRODUCTION .....	70
6.2 MERITS .....	70
6.2.1 Interface.....	70
6.2.2 Extension framework.....	70
6.2.3 Requirements .....	71
6.2.4 Completeness.....	71
6.3 SHORTCOMINGS .....	71
6.4 ENHANCEMENTS .....	72
6.4.1 Optimisation .....	72
6.4.2 Expressions.....	74
6.4.3 Indices and relational integrity .....	75
6.4.4 Portability.....	76
6.4.5 Functionality .....	76
6.4.6 Views .....	78
6.4.7 Null values.....	78
6.4.8 Data types.....	79
6.4.9 Caching .....	79
6.4.10 Data Dictionaries .....	79
6.4.11 Concurrency, recovery, and security.....	80
6.5 SUMMARY .....	80
<b>7. REFERENCES AND BIBLIOGRAPHY .....</b>	<b>82</b>
Appendix A - Technical documentation	
Appendix B - Test results	
Appendix C - Program code	

# Table of Figures

FIGURE 2-1 - GRAPHICAL REPRESENTATION OF A RELATION .....	4
FIGURE 3-2 - DATABASE/RELATION DATA STRUCTURE .....	8
FIGURE 3-3 - EXTERNAL HEADER STRUCTURE .....	12
FIGURE 3-4 - EXTERNAL BODY STRUCTURE .....	13
FIGURE 3-5 - EXTERNAL TUPLE REPRESENTATION .....	14
FIGURE 3-6 - CONCISE REPRESENTATION OF A TUPLE OF DEGREE $N$ .....	15
FIGURE 3-7 - GRAPHICAL REPRESENTATION OF THE DATABASE STRUCTURE.....	17
FIGURE 3-8 - OVERVIEW OF THE ALGEBRAIC OPERATORS .....	19
FIGURE 3-9 - PARSE TREE FOR AN EXPRESSION.....	28
FIGURE 4-1 - RELATION DATA STRUCTURE .....	44
FIGURE 4-2 - TUPLE DATA STRUCTURE .....	45
FIGURE 4-3 - GRAPHICAL REPRESENTATION OF A GENERIC STACK.....	47
FIGURE 4-4 - GRAPHICAL REPRESENTATION OF A HASH TABLE .....	49
FIGURE 4-5 - INDEX STRUCTURES .....	52
FIGURE 4-6 - ATTRIBUTE/HEADER FILE STRUCTURE .....	54
FIGURE 4-7 LEAP SYSTEM ABSTRACTION .....	56

# 1. Introduction

## 1.1 Basis and aims for the project

The field of databases is becoming an essential part of computing. Increasing numbers of businesses use database technology, from small single user systems such as Microsoft's Access, to larger multi-user distributed server systems such as Oracle and Ingres.

It is the relational theory that underlies the majority of such systems, for it successfully separates the user from dealing with secondary storage representation details (COD70), and has become the *de facto* standard. Courses in databases cover this model extensively for these very reasons.

Building upon the high level theory of the relational model<sup>1</sup>, the study of database management systems builds an understanding of the means by which the theoretical model is provided to users to use.

The *practice* of the theory underlying both the relational model, and relational database management systems, supports and enhances the students knowledge and understanding. It is only too true that there is insufficient time within the modular degree program for the student to build extensively on the theory, and to extensively practice the ideas.

This project aims to redress the issue, and build on the theory by designing and implementing a fully functioning relational database management system. However, it is simply not possible to cover all aspects of this large area. The design is therefore focused on the fundamental components of the relational model. The design includes a strong enhancement framework to enable the inevitable shortcomings to be redressed at a later date.

## 1.2 Document Structure

This document is split into six sections. This, the introduction, brings the project into focus and expands upon the reasons for the project being undertaken, introducing a rough scope. A more detailed scope is provided in section two, which formally defines a relational database management system. With the formal definition, the system design is then covered in section three, introducing the ideas that underlie the definition, and how they are to be approached within the implementation.

Issues that arise through the implementation of the specified design are covered in section four. Whilst not itself covering low level implementation issues relating to the coding, it discusses higher level issues such as the environment and language to be used, and approaches to specific problems. The testing of the system is covered in section five.

---

<sup>1</sup>And other models, i.e. Network and Hierarchical

Given that the project is the “design and implementation of a relational database management system”, the conclusion in section six assesses to what extent the project reaches this target. Particular merits of the system are highlighted, and some possible enhancements are discussed.

Appendix A contains a routine by routine description of the source code. Appendix B contains the source code itself, and Appendix C the results of the testing contained in section five.

In the interests of practicality, the project has been broken into several separately bound parts.

## 2. The scope of the system

### 2.1 Definition of a relational database management system

In order to be able to specify a system scope, a definition of a relational database management system is required. COD79 provides such a definition:

A database management system can be said to be fully relational if it supports three items:

- The structural aspects of the relational model
- The insert-update-delete rules (general integrity)
- A data sublanguage at least as powerful as relational algebra, even if all facilities the language may have for iterative loops and recursion were deleted from that language.

This definition is specific, but not detailed. The following sections deal with each item individually.

#### 2.1.1 The structural aspects of the relational model

COD79 defines a relation consisting of a set of tuples, with each tuple having the same set of attributes. If the domains<sup>2</sup> are all simple, such a relation has a tabular representation with the following properties:

The following definition of a relation should form the basis for the system's support of a relation.

- There is no duplication of tuples
- Row order is insignificant
- Attribute order is insignificant
- All table entries are atomic values

A relation is graphically represented in Figure 2-1.

---

<sup>2</sup>A domain is a set of values of similar type: for example, all possible part serial numbers for a given inventory or all possible dates for the class of events being record. A domain is *simple* if all its values are atomic (non-decomposable by the database management system). ( COD79)

## 2.1.2 The insert-update-delete rules

The modification of data already entered into the system is a task beyond the planned scope of the system, as the effort required is substantial. Briefly COD79 gives the rules as:

- 1) Entity integrity: No primary key value of a base relation is allowed to be null or to have a null component.
- 1) Referential integrity: Suppose an attribute  $A$  of a compound (i.e., multiattribute) primary key of a relation  $R$  is defined on a primary domain  $D$ . Then, at all times, for each value  $v$  of  $A$  in  $R$  there must exist a base relation (say  $S$ ) with a simple primary key (say  $B$ ) such that  $v$  occurs as a value of  $B$  in  $S$ .

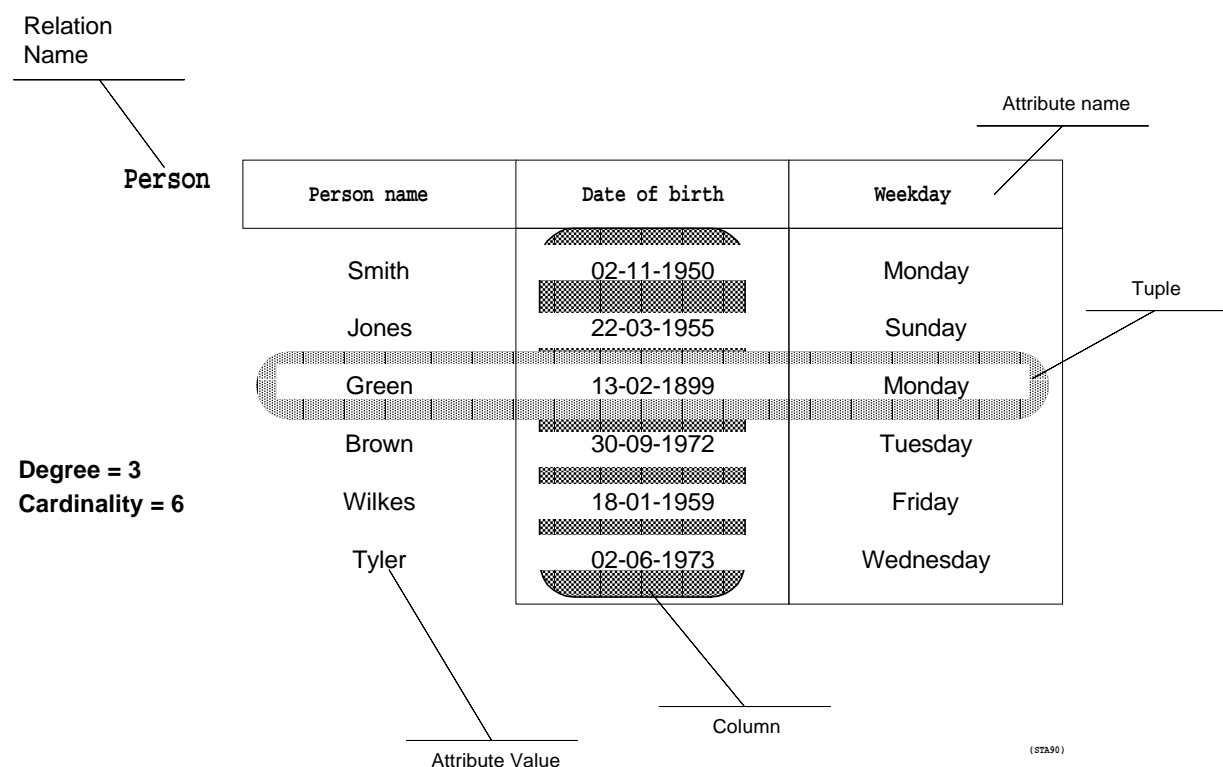


Figure 2-1 - Graphical representation of a relation

By not allowing, within the bounds of the system, the modification or deletion of data, restricting the system to querying operation only, these particular rules are not applicable. It is therefore true, that given all other components of a relational database management system, and an inability to modify data, this particular rule is to some extent enforced.

## 2.1.3 A data sub-language

The relational algebra is a simple, explicit and well defined query language. Other sub-languages exist, e.g. SQL, and QUEL. However, these are extensive systems which go much further than the relational algebra in terms of functionality. Their

implementation is a project in itself, and clearly exceeding the scope of this present project.

## **2.2 Project scope**

A fully relational system should provide the functionality describe by the definition of a relational database. In the current context some limitation is necessary, therefore the project scope includes:

- Support for the structural aspects of the relational model
- A sizeable subset of the relational algebra
- A well-defined framework for further enhancement.

The system, to be referred to as LEAP<sup>3</sup>, should cover offer this criteria.

---

<sup>3</sup>LEAP - Leyton Extendible Algebraic Processor

## 3. System Design

### 3.1 Introduction

This section will cover in detail the conceptual framework on which the implementation of LEAP will be based. A high level approach is taken. The design is *not* intended to be a formalised and ‘cast in stone’. Rather a loose framework is given which should form a clear foundation for technical development.

The design is therefore rather abstract, and the implementation reflects this. As such, this section considers the low-level structural concerns of the relational model, before moving to the operators which utilise these structures.

The highest level, that of the user interface, is considered last. It gives end-users full access to the systems functionality.

The enhancement framework spans across all levels of abstraction, and is therefore dealt with last of all, such that previous sections may be referenced and drawn upon.

### 3.2 The design of the structural aspects of the relational model

#### 3.2.1 Introduction

A definition of the structures that constitute the relational model is required. The definition given by DAT90 is concise, and provides a good start point in determining the complete set of structures.

A relational database is said to be “*a database that is perceived by the user as a collection of time varying, normalised relations of assorted degrees*”.

This is not in itself complete. Further analysis and definition gives:

**A database contains a collection of relations** - Many databases may exist, but a relation relates to only one database<sup>4</sup>. Some provision is therefore necessary to create the database structure itself. This will contain a number of relations. Through such a structure, provision is immediately available for multiple database structures within the one system.

**Relations** - Relations are the core structure of the relational database. A relation consists of two parts, a “header” and a “body” (DAT90).

**Header** - The heading consists of a fixed set of attribute/domain pairs:

---

<sup>4</sup>Although some database systems allow a relation to be made widely available to other databases, database systems etc. To achieve a truly distributed structure.

$$\{ (A1:D1), (A2:D2), \dots, (An:Dn) \}$$

such that each attribute  $A_j$  corresponds to exactly one of the underlying domains  $D_j$  ( $j = 1, 2, \dots, n$ ). (DAT90).

**Body** - The body consists of a time-varying set of tuples, where each tuple in turn consists of a set of attribute-value pairs:

$$\{ (A1:vi1), (A2:vi2), \dots, (An:vin) \}$$

( $i = 1, 2, \dots, m$ , where  $m$  is the number of tuples in the set).

In each such tuple, there is one such attribute-value pair ( $A_j:vi_j$ ) for each attribute  $A_j$  in the heading. For any given attribute-value pair ( $A_j:vi_j$ ),  $vi_j$  is a value from the unique domain  $D_j$  that is associated with the attribute  $A_j$ . (DAT90).

Each of these structures must be implemented if the structural aspects of the relational model are to be provided. The following sub-sections take each in turn, and discuss abstractly the means (data structures & operations) by which the system will provide them.

### 3.2.2 The database structure

The database is composed of relations. As a number of databases may conceivably exist within the system, the database will also need some unique identifier, such as a name.

Relations must be tied to a database by some means, to enable a specific relation to be located and processed. Some structure is necessary to facilitate this. Therefore, the database structure should contain:

- Name
- Attached relations

#### 3.2.2.1 Data structures

The data type for the database structure can be a simple record, which contains the specified data. By referencing this structure, a routine will be able to locate the necessary information.

The attached relations must be structured in some way (The structure of relations themselves is discussed in section 3.2.3). The requirements for this structure are:

- The structure should contain all relations that have been inserted into the structure.
- The structure should facilitate searching for a given relation.

- The structure should allow relations to be added and removed.

The structure need not be complex. The number of relations that will reside within the structure will not be large - tens rather than hundreds. The structure will certainly be searched frequently, but not excessively. Relations should only need to be located once per algebraic operation.

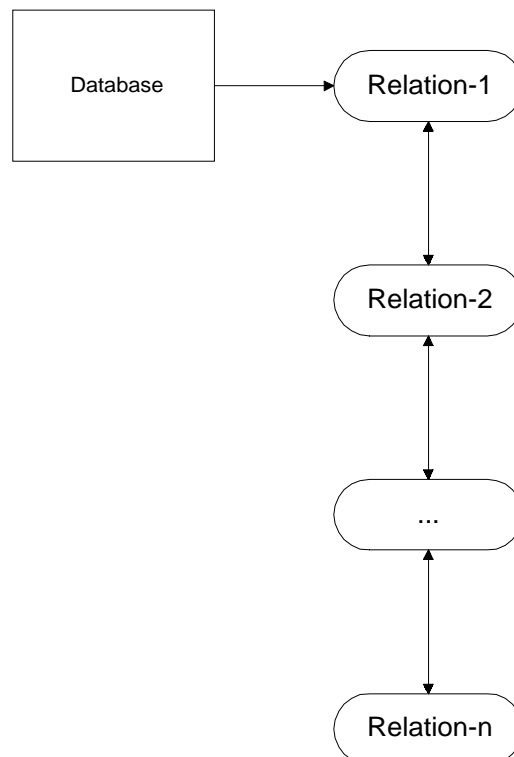
Given these requirements, and an understanding of the use to which the structure will be put, two possibilities exist for the structure: A binary search tree and an ordered linked list. Both satisfy the requirements, and offer advantages and disadvantages.

The binary search tree is fast for retrieval, and insertion is not overtly complex. Deletion is more complicated and time consuming.

The ordered linked list is slower than the binary search tree for retrieval and insertion, when there exists a large number of entries. Deletion is less complex, and quicker.

The ordered linked list is the appropriate operation in this scenario. Given the estimated size of the structure that will exist, a binary search tree would exceed the requirements.

A diagram giving a graphical representation of the data structures discussed is given in Figure 3-2.



**Figure 3-2 - Database/relation data structure**

### 3.2.2.2 Operations

There are five operations on the stand-alone database structure. A database must be explicitly created. If a higher level structure containing a number of databases were conceived, the databases would be added to this higher level structure as appropriate. The database should also be destroyed when the system terminates.

The insertion, deletion and searching of the relation structure is also necessary to add, remove and locate relations respectively (the creation of relations is discussed in section 3.2.3)

Therefore, the following operations are required:

Operation	Description
<b>Create database</b>	A database of a specified name should be created, which will contain specific relations at a later date.
<b>Destroy database</b>	The specified database should be destroyed.
<b>Insert relation</b>	A specified relation should be inserted into the database relation structure at the correct location, maintaining the structure of the list.
<b>Delete relation</b>	The specified relation should be removed from the database relation structure, and the structure of the list maintained.
<b>Locate relation</b>	A given relation should be located within the database.

### 3.2.3 The relation structure

Section 3.2.2 discussed how the database structure itself should be implemented, whilst this section discusses the implementation of the relation structure. A relation on a high level is composed of the header and body structure. Structures are discussed in section 3.2.4, and section 3.2.5. They are not covered in significant detail in this section.

#### 3.2.3.1 Data structures

A great deal of meta-information may be stored with respect to every relation. The most important of these are the heading and body structures. References to these structures are clearly necessary.

However, beyond noting that such reference information is stored in these structures, they are not of additional concern at this stage. There is other, meta-information that may be stored with regard to a relation:

- The relations name
- The status of the relation
- Index information

- Creation details
- Ownership information

Clearly not all such information is necessary in the LEAP system. For example, ownership information is only applicable in large multi-user systems.

The only information that need be specifically stored within the relation structure is name and status since relations must be uniquely identifiable. The naming of a relation should on the whole be controlled by the user, who is likely to wish a core relation to have a name that relates it in some way to the data it contains.

A relation may be created by the system itself as a result of executing some commands. Such relations may not be meaningful outside the bounds of the operations. Therefore these relations would need some ‘flag’ indicating a temporary status to enable them to be purged at some point. Without such a flag, all relations are of equal status, and the system may become cluttered with unnecessary structures.

Index information is outside the bounds of the core relation structure. Indices provide a faster means to access a relation. They are discussed in more detail in section 4.5.4.6.

Creation details are, like ownership information, only necessary in systems where a large number of relations exist. Large databases may necessitate the need for relations to be tracked, and a log of relation creation maintained. This is not necessary within the development of LEAP, and certainly does not come under the core relation structure.

There is no need for additional data structures within the relation. The maintenance of relations is managed by the operations discussed in section 3.2.2.2. The form of data contained within the relations is discussed in the following sections.

### 3.2.3.2 Operations

Operations are still necessary to work on the `relation` data type. The structures must be created, disposed of, and managed.

Operation	Description
<b>Create relation</b>	Creates a specific relation.
<b>Delete Relation</b>	Disposes of a specific relation
<b>Erase Relation</b>	Removes a relation permanently
<b>Relation Name</b>	Returns the name of a relation

Relations must be stored in some permanent form. The higher level relation structure can be derived from the secondary storage structures for the heading and body, if a standard naming convention for the files is used.

### 3.2.4 The heading structure

The relation structure discussed in section 3.2.3 contains a reference to the structure of the relations heading. The heading defines the attribute/domain pairs.

#### 3.2.4.1 Data structures

The heading will therefore consist of:

- Attribute
- Domain

The entire domain cannot feasibly be stored as it is primarily conceptual in nature, therefore some restriction is appropriate. A number of defined data types should be used, which implies the possible values to some extent. Examples of this are:

- String
- Number
- Date

Whilst this does not allow a particularly accurate mapping of a real world domain *per se*, a number of attribute/domain pairs can be grouped by the user, within a relation, if necessary. This is the approach traditionally taken by RDBMS. User defined types are becoming available, in systems such as Oracle 7. Clearly such a system is outside of the scope of a project of LEAP's scale.

The attribute may be stored simply, as it consists of a name which relates the data contained within the appropriate pairing of the body to the real world.

A discussion is required as to the storage of the heading at this stage. Without prejudice to implementation details, there is a clear requirement for the heading to be stored on some form of permanent secondary storage medium, as well as an internal representation.

The permanent storage structure must contain sufficient information to enable an identical internal representation to be created.

In terms of data content, the differences between the heading and the body imply two separate structures are appropriate. The structure for the heading must store the attribute and domain. A simple structure would list the attribute name and the domain/data type sequentially. The end of the structure would indicate there are no more attribute/domain pairs (See Figure 3-3).

### 3.2.4.2 Operations

Should the internal representation of attributes facilitate an independent structure, e.g. a linked list of attributes? The use to which attributes *themselves* will be put must be answered.

Attributes will be used for the location of a particular set of values within a relation, which occurs only when tuples are processed. Each tuple must contain some reference to the attributes. There is no foreseeable necessity for attributes to be accessed independently of a tuple. There is, therefore, no anticipated need for attributes to have a data structure of their own.

Externally the attributes must be accessed appropriately, and the provided operations should enable this. Attributes should be located and loaded into an internal representation.

Operation	Description
create attribute	Create an attribute for a specific relation
attribute findfirst	Locate the first attribute of a relations header
attribute findnext	Locate the next attribute in a relations header
attribute name	Get the name of an attribute
attribute type	Get the type/domain of an attribute
attribute dispose	Dispose of an attribute.

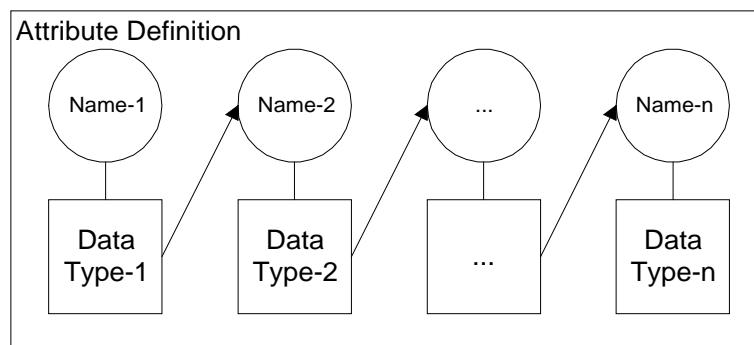


Figure 3-3 - External header structure

### 3.2.5 The body structure

The accepted description of a tuple implies the necessary structure, where there exists a direct cross-reference between a tuple data value, and the attribute to which it relates. The tuple values themselves will, by definition, vary with each distinct tuple in a given relation. The attribute values will not.

#### 3.2.5.1 Data structures

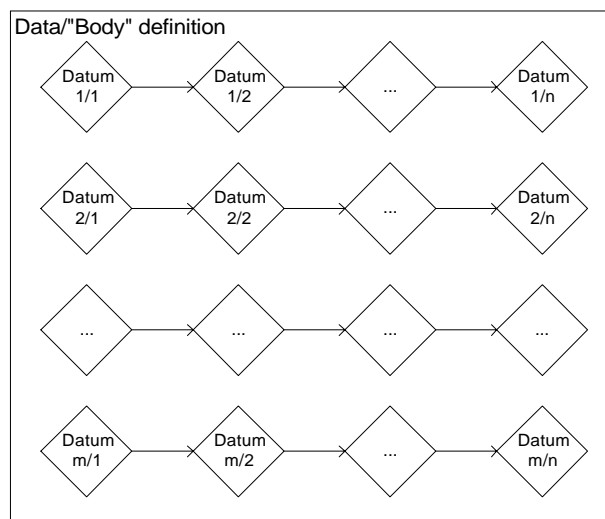
Tuples must be self-contained, and hold to the definition given above. An external representation of tuples may relate to the heading storage structure. Internally, the representation must contain a reference to an internal attribute structure. Referencing an external structure would be slow and laborious, and would necessitate repeated accessing of the attribute structure. This is not necessary.

##### 3.2.5.1.1 External Structure

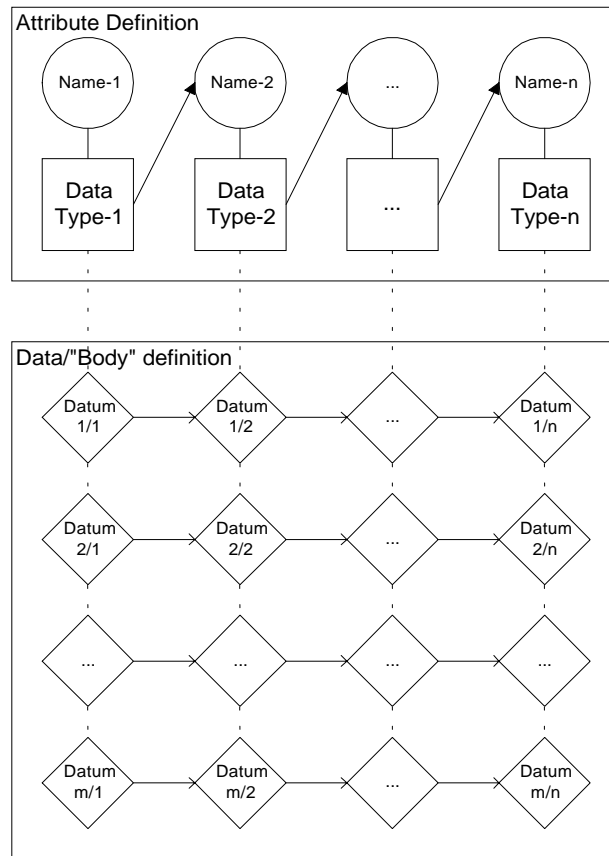
The external structure of a tuple will have a strong correspondence with the external structure of the heading. This was specified as an ordered structure of attribute name, and domain/data type. The tuple values themselves will need to relate to this list of attributes.

The most appropriate method is structuring the values in identical order to the attribute/domain structure. Attributes and values are thereby linked by structure location. If attribute values were stored in some other ordering, a specific reference to the attribute/domain value would be required for each tuple value, in every tuple. This would be expensive in both space and processing terms.

Figure 3-4 gives a graphical representation of an appropriate structure. Figure 3-5 demonstrates the inter-relation of the header and body sections of the relation.



**Figure 3-4 - External body structure**



**Figure 3-5 - External tuple representation**

### 3.2.5.1.2 Internal representation

The internal representation of tuples must be self contained. It would be expensive in processing and I/O if the external structures were accessed repeatedly, rather than maintaining a complete internal representation. Within an operation, any number of tuples will be necessary at any one time. Binary<sup>5</sup> operations especially will require two tuples from different relations, and nested expressions may possibly require additional structures.

Both attribute and value needs to be available for an operation.

The tuple will therefore consist of at least:

- Attribute/Domain definitions
- Values

The relationship between value and attribute *must* be maintained for the data to remain valid. Figure 3-6 gives a concise representation of a tuple of degree  $n$ .

---

<sup>5</sup>Binary, as opposed to unary operations, in relational algebra terms.

Attribute 1	Attribute 2	...	Attribute $n$
Value 1	Value 2	...	Value $n$

**Figure 3-6 - Concise representation of a tuple of degree  $n$**

There are two methods by which the tuple may be implemented: A dynamic linked list, or a static array. Other structures, such as binary search trees could be utilised, but their structural differences would add to the conceptual complexity of LEAP, without bringing about any benefits.

A dynamic linked list would clearly be more efficient in terms of memory, yet it would be cumbersome to access a particular element - the entire linked list would have to be searched to locate an attribute and value. A static array, whilst not as efficient with memory<sup>6</sup>, would be far more dynamic. Arrays are an integral part of most high level programming languages, and can be accessed quickly within the bounds of the language without recourse to abstraction routines.

The static array is therefore the appropriate option.

### 3.2.5.2 Operations

The body data structures are different and distinct. The operations that operate on them are also distinct.

#### 3.2.5.2.1 External structure operations

The operations that are applicable on the external structure should locate a tuple, and build an internal structure. There will be utilisation of the header operations in the process.

Tuples are not stored in any particular order within a relation. Should indexing be designed/implemented at some stage, specific tuples will need to be located. The operations should offer this facility.

Operations to read tuples should, where appropriate, build the internal representation of the tuple. Operations to write tuples should add the tuple to the external structure in the appropriate location (normally the end, as ordering is not significant). Indices may need to be updated, in which case write operations should return some indicator to the location at which the tuple was written, to enable a direct read at a later date.

---

<sup>6</sup>Because arrays are usually of fixed size, a maximum size for a relations degree would have to be specified. All tuple representations would then have to be of this size, regardless of their degree.

Operation	Description
<b>write tuple</b>	Write a specific tuple to the appropriate file.
<b>Read first tuple</b>	Read the very first tuple from the appropriate file
<b>Read next tuple</b>	Read the next tuple from the appropriate file

### 3.2.5.2.2 Internal structure operations

Once an internal representation of a tuple has been built, some means to access this data is necessary. An array representation of the tuple enables the direct accessing of elements, but further facilities will be necessary. For example, a tuple may have been built. A given attribute must be located within the tuple, to determine the position of data within the structure.

A tuple must be built at some stage to enable output from a routine to be written to disk. A routine to create an 'empty' tuple that the system may then populate is necessary.

Operation	Description
<b>Build tuple</b>	Create a new empty tuple, based on a given relation.
<b>Find attribute</b>	Locate an attribute in a tuple.

## 3.2.6 Summary

All structures that constitute the relational model have been discussed. The Inter-relation of structures has only been briefly discussed. This section will cover this in more detail.

- The database structure contains a reference to a list of relations. All relations within a database should be present in this structure.
- A relation contains information about itself, on a high level, i.e. its name and status. References to the external structures that constitute the structure of the relation are stored. The header and body are treated separately.
- The header contains the attribute/domain pairs of the relation in a sequential format. There is no internal data structure for a header element, as they are held independently within tuples as appropriate.
- The body contains the tuples - attribute values are paired with the appropriate attribute in the header through ordering. Internally, the tuple consists of attribute/value pairs represented in an array (See Figure 3-6).
- The internal/external structure is graphically represented in Figure 3-7.

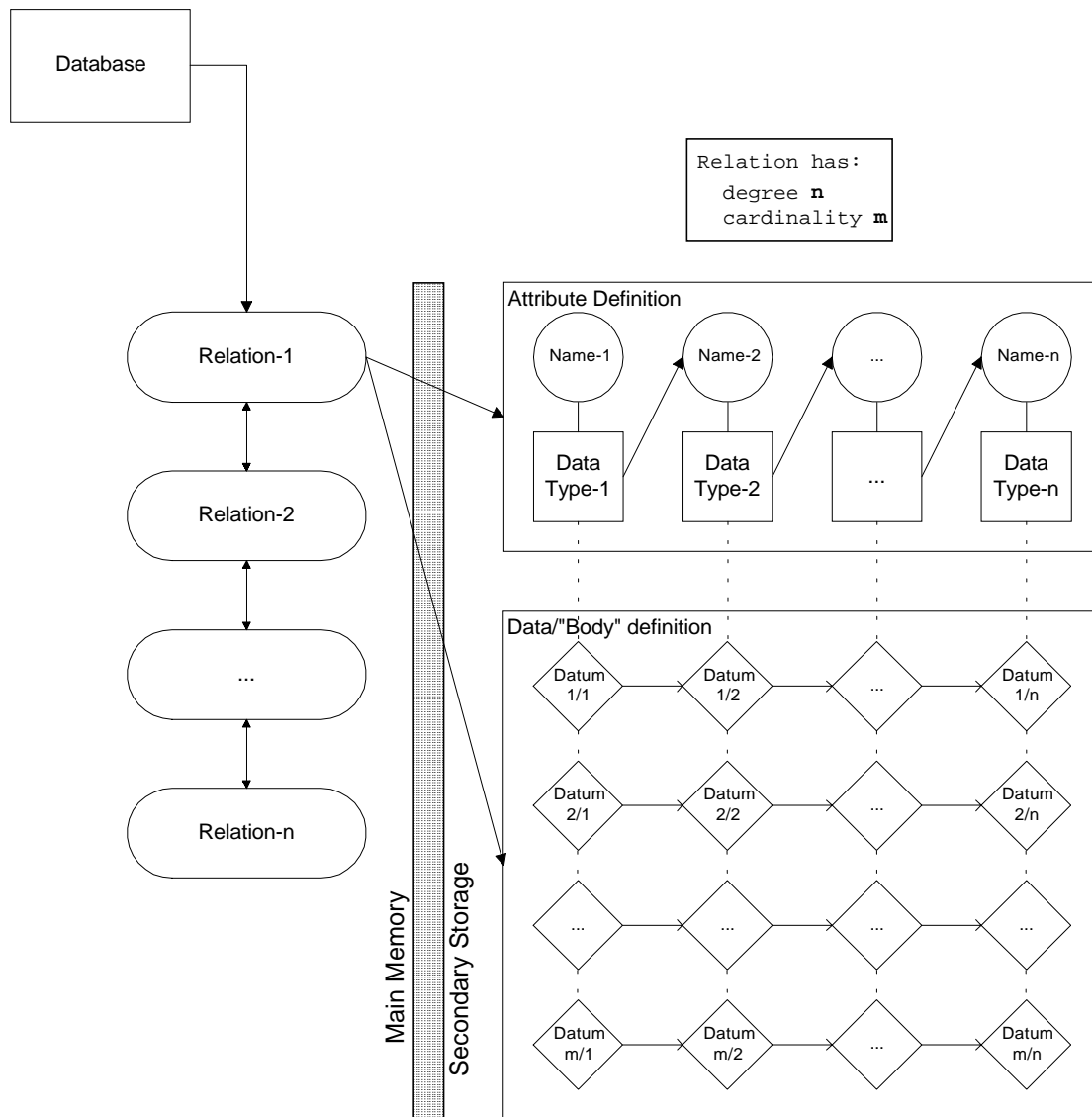


Figure 3-7 - Graphical representation of the database structure

### 3.3 The data sub-language

#### 3.3.1 Introduction

The structural aspects of the relational model form the means by which the data is stored within the system. In order to *extract* meaningful data, a set of operators is necessary. The relational algebra is an elementary language which provides the core functionality any relational database management system must provide.

There were eight original operators specified by E.F. Codd (cited in DAT90):

**Project:** Extracts specified attributes from a specified relation (section 3.3.3).

**Union:** Builds a relation consisting of all tuples appearing in either or both or two specified relations (section 3.3.4).

- Intersect:** Builds a relation consisting of all tuples appearing in both of two specified relations (section 3.3.5).
- Difference:** Builds a relation consisting of all tuples appearing in the first, *but not* the second of two specified relations (section 3.3.6).
- Product:** Builds a relation from two specified relations consisting of all possible combinations of tuples, one from each of the two relations (section 3.3.7).
- Join:** Builds a relation from two specified relations consisting of all possible combinations of tuples, one from each of the two relations, such that the two tuples contributing to any given combination satisfy some specified condition (section 3.3.8).
- Select**<sup>7</sup>: Extracts specified tuples from a specified relation (section 3.3.9).
- Divide**<sup>8</sup>: Takes two relations, one binary and one unary, and builds a relation consisting of all values of one attribute of the binary relation that match (in the other attribute) all values in the unary relation.

Figure 3-8 gives a graphical overview of the operators.

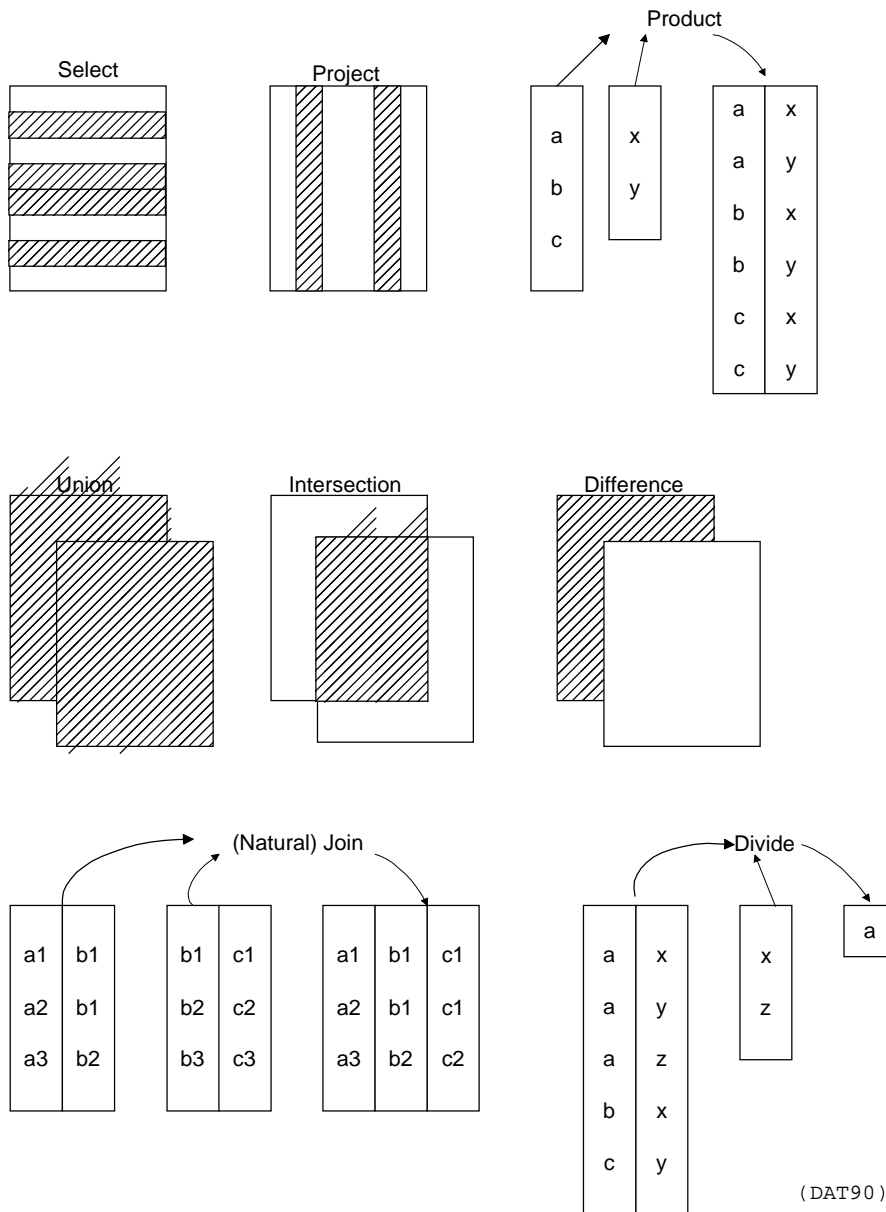
---

<sup>7</sup>**Restrict** was the original name for this operation, but select is more commonly used.

<sup>8</sup>Divide can be expressed via a combination of other operators: Given  $r(R)$  and  $s(S)$  with  $R \supseteq S$  and let  $R' = R - S$ :

$$r \div s = \pi_{R'}(r) - \pi_{R'}((\pi_{R'}(r) \bowtie s) - r)$$

(MAI83). Therefore, it has not been implemented (See testing for LEAP example of this notation)



**Figure 3-8 - Overview of the algebraic operators**

Additional functions have been suggested by other writers. These include: **Extend**, which enables computation to be made on attribute values; **Summarise**, which enables the summarisation of groups of attribute values; and others.

LEAP will aim to provide a subset of the original operators. The design of these operators is discussed in this section. The implementation aspects are discussed later (section 4). The operations discussed in section 3.2, particularly those for the heading (section 3.2.4) and body (section 3.2.5) structures of relations will be of direct use, for the creation and population of relations.

### 3.3.2 General requirements

Each of the operators must support the principle of relational closure, whereby any algebraic operation yields a relation as a result. The ‘source’ relations must not be altered in any way, and the resulting relations should be valid relations, with no duplicate tuples.

### 3.3.3 Project

The project operator yields a relation that contains only the specified attributes of the operand.

#### *Method*

The operator will take as an input one source relation, and a list of attributes. The specified attributes and their values within the relation will be output to the resulting relation.

The output relation should first be created. Each specified attribute should be located within the source relation, and created in the destination relation with the same data type/underlying domain. Once the destination relation has been created, tuples from the source relation should be read sequentially.

The data contained in the source tuples should be copied, as appropriate, to a tuple for the output relation. A check should be performed using some appropriate technique, to determine whether the tuple already exists in the destination relation (As attribute-values are removed, duplicate tuples become possible). If no such tuple exists, the tuple should then be written to the destination relation.

An alternative approach would be to create a duplicate relation, and remove that specified attributes, then remove duplicate tuples. This approach would be more expensive, as a read/write would be required for each tuple when creating the duplicate. This approach is mentioned in books such as MAI 83, as the conceptual approach.

#### *Optimisation*

There is no way by which the operator can be prevented from having to process each tuple within the source relation.

It is the area in which tuples are read from the source relation and written to the destination relation, that most work is performed. This therefore is the appropriate point for identifying an optimal algorithm.

The reading and writing of tuples cannot be optimised. They are to be implemented atomically, with the minimal number of different read/write operations. Therefore, the process by which the tuple is checked for existence within the destination relation is the appropriate position for optimisation within *all* operations. A number of methods are available:

- Search:** The entire destination relation can be searched prior to a tuple write. If a duplicate is located, the write should be aborted and the next tuple retrieved.
- Hash:** A hash table is maintained of all tuples written to the destination relation. Prior to a write, the hash table is queried. If an entry is located within the hash table, the tuple is not written. The tuple itself would be utilised as the hash key.
- Index:** Prior to a tuple write, the tuple is searched for within index structures that are created with the new relation. If the tuple already exists, it is not written, otherwise the tuple is written, updating the index structure.

All approaches specified here are feasible, but different in terms of operational speed and complexity. The hashing approach is the most appropriate due to its relative simplicity and clear speed advantage - certainly over that for search. In comparison to the index approach, the hashing approach is likely to be at least as fast, but far simpler. This methodology should be implemented.

### 3.3.4 Union

The union operator yields a relation containing all tuples appearing in either or both of two specified *union-compatible* relations. A relation is said to be union-compatible if it contains the same set of attributes, and that all corresponding attributes are defined on the same domain.

#### *Method*

The operator will take two source relations. A check is necessary for union compatibility, which should abort the operation if the source relations are not compatible.

Given that the source relations are union compatible, the output process should commence. The tuples from the first relation should be output to the destination relation (which will be union compatible with both source relations) without any checks. The tuples from the second relation should be output to the destination relation following a check on the output relation, to ensure that no duplicate is written.

#### *Optimisation*

This particular operation is very straight forward in concept and method. There is little room for alternative methods and optimisation.

Optimisation techniques are identical to those discussed in section 3.3.3 - only where the check is made for a duplicate tuple does there exist room for improvement. All tuples must be processed once in both relations at some point. There is no means by which the tuples of one relation may be skipped without first reading and processing them.

Given that the hash methodology is appropriate for duplicate determination within the `project` operation, it follows that it will also be the appropriate operation in the same circumstances elsewhere.

### 3.3.5 Intersect

The intersect operator yields a relation with the same attributes as its source relations (see the `union` operation). The relation contains all tuples that are present in both of the source relations. Source relations must be *union-compatible*.

#### *Method*

Taking two source relations, a check should be made for union compatibility which will abort the operation should it fail.

Assuming the relations are union compatible, the output relation should then be created. The output relation will be union compatible with both source relations. A tuple is read from the current source relation, and a check should be made to determine whether it exists within the other source relation.

If the tuple does exist in the other relation, it should be written to the destination relation. The process should read all tuples from both relations, and check the corresponding relation before each write.

#### *Optimisation*

The check for duplicate tuples in the corresponding relation is the only point at which optimisation might occur. Methods are discussed in section 3.3.3. A hash table is again the most appropriate method. The first relation may be read, and the hash table populated with the tuples. No tuple write need occur at this stage.

The second relation's tuples should be compared against the hash table. If a tuple is found to exist in the hash table, and therefore in the first relation, the tuple may be written. No further addition to the hash table should occur.

### 3.3.6 Difference

The difference operator yields a relation which is union compatible with its source relations. The yielded relation contains all tuples that belong to the first relation, *but not* the second.

#### *Method*

Taking two source relations, a check should be made for union compatibility which should abort the operation should it fail.

Given that the relations are union compatible, the output relation should then be created. The output relation will be union compatible with both source relations. A

tuple is read from the current source relation, and a check should be made to determine whether it exists within the other source relation.

If the tuple does not exist in the other relation, it should be written to the destination relation. The process should read all tuples from the first relation, cross-checking the second relation entries.

### *Optimisation*

The difference operator is straight forward, but there is room for optimisation. In the above method, the second relation has to be searched each time. Potentially this could require the relation to be processed a very large number of times.

By using hashing, and reading the second relation in only once in order to build the hash table, a significant speed advantage is realised. A check can then be made against the hash table to determine if a tuple is present in the second relation, before a tuple write operation is executed.

Indexing would remove the necessity for a hash table, and its necessity to read the entire relation. Assuming an index exists, this could be searched for a specific tuple.

## **3.3.7 Product**

The product operator yields a relation which is the Cartesian product of two source relations. That is, all possible combinations of the two source relations. The resulting relations contains all attributes of the source relation.

### *Method*

Each tuple within the first relation should be combined with all successive relations in the second relation and written to the resulting relation. There is no necessity for union compatibility at all.

### *Optimisation*

All tuples from the first relation are combined with all tuples from the second relation. This operation is clearly repetitive, but there is no chance of a duplicate in the destination relation<sup>9</sup>. The caching of a relation, or part of a relation is possible to reduce the number of expensive disk read/writes that will occur.

This approach could possibly be expensive in terms of memory, but would significantly improve the speed of the system. However, in the current implementation the raw reading and writing from/to disk is appropriate, despite its expense.

---

<sup>9</sup>Both source relations have no internal duplicates by definition in COD79, there will therefore be no duplicate in the resulting relation when calculating the PRODUCT.

### 3.3.8 Join

The join operator yields a relation which is the Cartesian product of two source relations, with some qualification applied to the tuples that compose the yielded relation. Together with project, it forms the basis for relation normalisation.

Varying kinds of join exist, e.g. *equi* join, *natural* join etc. which contain slightly different attributes. The equi join (which contains all attributes from both source relations) will be discussed, although the principles may be applied to the other joins<sup>10</sup>.

#### *Method*

The join operation is essential in a queries involving multiple relations. Because of this, it is implemented separately to ensure a more efficient operation, despite the fact that a Cartesian product followed by a `select` (and possible a `project`) produces the same result. A number of methods exist:

#### 3.3.8.1.1 Nested loop

A tuple is taken from the first relation. This tuple is then compared with all tuples from the second relation. If the join condition is satisfied, the joined tuple is written out. Once all tuples have been read from the second relation, the next tuple is taken from the first tuple, and the process repeated.

#### 3.3.8.1.2 Sort and Merge method

This method requires both relations to be sorted, and requires only one pass of both relations. A group of tuples of one relation with the same value on the join attributes is read, then the corresponding tuples (if any) of the other relations are read. Since the relations are in sorted order, tuples with the same value on the join attributes are in consecutive order. This allows each tuple to be read just once. (KOR91).

#### 3.3.8.1.3 Hash join

A hashing table is constructed on one relation, and the second relation is processed. If a clash occurs when looking up the hash value in the hash table for the second relation, a check is made to ensure that the join attributes are equal. If so the appropriate join tuple is created and written (David .J. DeWitt et al. Cited in DAT90).

#### 3.3.8.1.4 Conclusion

The nested loop algorithm is simple and flexible, and does not require additional complexity<sup>11</sup>. This should therefore be implemented, with attempts being made if time permits to implement others.

---

<sup>10</sup>The natural join can be created by applying a `project` to the result of the equi-join. By not disposing of any data, the onus is with the user to resolve repetition.

<sup>11</sup>For example, in the resolution of expressions, i.e. the normal routine breaks down with non-equal expressions.

### *Optimisation*

The above methods are distinct, and different approaches to the problems. They offer advantages and disadvantages. Different RDBMS implementations adopt different techniques. Hashing is often utilised because sorting is not necessary, but it is expensive in terms of memory. An advanced RDBMS system will possibly contain a number of algorithms, and utilise whichever is most appropriate in the current scenario.

LEAP will, because of the inevitable restrictions imposed by time, implement the simplest, with more if time permits.

### **3.3.9 Select**

The select operator yields a relation containing only tuples from a specified relation, that satisfy a given condition.

#### *Method*

Each tuple from the source relation must be read, and the given condition evaluated with the current tuples attribute-values. Tuples which satisfy this condition should be written to the source relation.

In a similar manner to the `project` operation, a duplicate tuple could be created, and tuples not matching the condition removed from the new relation. As with the `project` method briefly discussed, this is an expensive approach, and not appropriate.

#### *Optimisation*

The condition will determine which tuples should be written to the destination relation. Without optimisation, each tuple from the source relation will have to be read and evaluated.

Indexing may be utilised to quickly locate those tuples which have the given attribute values. However the number of possibilities within conditions, especially when combined to form larger conditions, may ensure that the optimisation is complicated.

It is not possible at this stage to determine an index based approach given that it is not part of the core requirements. Therefore, the non-optimal approach is to be used.

## **3.4 The interface**

### **3.4.1 Introduction**

The user interface provides the means by which the user may interact with the system. It should enable all of the functionality of the underlying system to be exploited and utilised in an easy and predictable manner.

There are a number of approaches that may be taken:

**A command based interface** - whereby the user enters commands, which the system interprets and acts upon, producing a result.

**A graphical based interface** - whereby the user can enter commands, but using a graphical means as opposed to commands.

**A unit interface** - whereby the user can incorporate the work of the database system, but within their own program(s).

In the proposed implementation environment, a graphical user interface would involve extensive additional work. It is not simply a matter of producing a graphical user interface - the interface must be fluid and intuitive. The facilities provided by the Microsoft Windows environment make for simpler development, but extensive thought and consideration is still nonetheless required.

Relational algebra is by its very nature geared toward a procedural approach which matches well with a command line interface. RDBMS query tools in large commercial systems were primarily command line based (such as Oracle's SQL), and are only now moving towards a graphical front end environment with graphical representations of relations and queries (e.g. Oracle Forms; Business Objects etc.).

A unit interface would be expected regardless of the actual interface provided as a part of the dissertation. A number of units may be imported within the interface program, which would provide the functionality required of the relational database management system.

### 3.4.2 Language

Before any decisions can be made with regard to the actual design of the interface, a decision must be made with regards to the language to be provided.

The decision is straight forward, given the size and resources available. The project scope specified that a core subset of the relational algebra functionality should be made available.

Yet SQL and QUEL all provide at least the functionality of relational algebra. They must do, following COD79's definition of a relational database management system, if they are to be utilised as part of a larger relational database offering.

It is clearly not feasible within LEAP to implement a standardised query language such as SQL. This is a large language with a detailed international specification, but there are means by which a language interpreter can be added to an engine such as discussed in earlier sections.

UNIX provides an engine such as discussed in earlier sections. UNIX provides the *Yacc* and *Bison* tools. If such an environment were being utilised, and the appropriate languages used (C is necessary to utilised both of these tools), then it would be viable to implement at least a sizeable subset of SQL or other formally specified language.

The tools would take care of the parsing of the command, and build an internal representation of the query.

However, these tools are not available. Therefore the possibility of an SQL interpreter must remain an idea alone in this project. Scanning the relational algebra sections of books such as DAT90, DES90, and STA90 yields the answer.

These books discuss the relational algebra, and provide an imaginary language based entirely upon the relational algebra operators. Different notation is used, but the principle is identical.

This approach is simple, and clear. It is therefore the appropriate ‘language’ to use for LEAP. STA90’s approach is the most legible, utilising the names of algebraic operators directly, rather than DES90’s approach utilising the commonly used algebraic symbols (project -  $\pi$ ; select -  $\sigma$ ; etc.). Both are semantically identical.

### 3.4.3 Syntax

A formal definition of the proposed language would be appropriate at this point. The following is in Backus-Naur-Form (BNF):

```
<command> ::= [ <relation name> = ] <relational-exp>
<relation-exp> ::= <unary-term> | <binary-term> | relation-name
<unary-term> ::= project((<relational-exp>)(<attribute-list>)) |
                 select((<relational-exp>)(<qualification>))
<attribute-list> ::= <attribute-name> [ , <attribute-name> ]*
<qualification> ::= <attribute-comp> [ { and | or } (<attribute-comp>)]*
<binary-term> ::= <operator>((<relational-exp>), (<relational-exp>)) |
                 join((<relational-exp>), (<relational-exp>), (<qualification>))
<operator> ::= union | diff | intersect | product
<attribute-comp> ::= <attribute-name> <comparator> { attrib-name | value }
<comparator> ::= { < | > | <= | >= | = }
```

### 3.4.4 Additional Functionality

The interpreter will provide a subset of the relational algebra, in addition to this a number of additional features will be necessary:

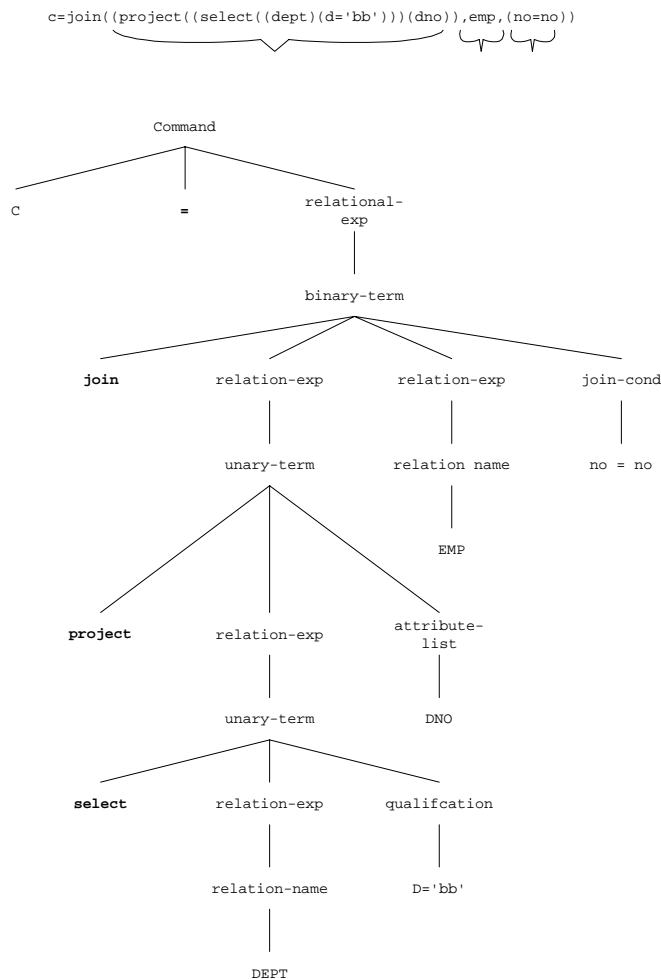
- **Maintenance** - The relational structures will need to be added, deleted, and modified.

- **Inspection** - The user will want to be able to inspect the result of algebraic operations. A number of operations will enable them to view relational structures as created whilst an expression is processed.

These do not form a part of the BNF definition of the language for they do not produce anything. Rather they offer a service to the user to facilitate the process of data querying.

### 3.4.5 Approach

Given the BNF description of the language, when the program executes, a command will be processed in the appropriate order. See Figure 3-9.



**Figure 3-9 - Parse tree for an expression**

The implementation of the BNF description may vary in the technique, but the result should be the same.

Briefly there are two key approaches to processing a command: A two stage parse/execute approach; and a one stage integrated parse/execute. The first approach will enable query optimisation (critical in large database systems), but with added complexity; the second is simpler, but harder to debug.

The second approach is the appropriate option in the current system. A greater discussion of this is given in the section on Command processing (section 4.5.3)

### 3.4.6 Syntax transformation

The syntax provided in the relational algebra section of DAT90 is in infix form, with the operator sandwiched by the relations to which it applies, in the same way conventional mathematical expressions are expressed:

$$4 \times 5$$

The BNF language define uses prefix form, which for the above expression is equal to:

$$\times 4 5$$

The operator appears before the operands. With nesting, e.g.

$$(4 \times 5) + 3$$

in prefix form, is:

$$+ (\times 4 5) 3$$

An infix algebraic form, for example:

$$A \text{ MINUS}^{12} B$$

when expressed in prefix form, as given in the above BNF language, becomes:

$$\text{DIFFERENCE} ( ( A ) ( B ) )$$

and nesting:

$$\text{PROJECT } s\#, \text{ STATUS, CITY } ( \underline{A \text{ MINUS } B} )$$

becomes:

$$\text{PROJECT} ( ( \underline{\text{DIFFERENCE} ( ( A ) ( B ) ) } ) ( S\#, \text{ STATUS, CITY } ) )$$

The infix expression may be read as “*project attributes s#, status, and city from the relation that results from a minus b*”. The postfix expression may be read as “*project, from the result of executing the difference between a and b, the attributes s#, status and city*”.

A prefix notation is a simpler approach to implement, and is not additionally complicated for the reader to understand.

---

<sup>12</sup>MINUS is used in DAT90 as the DIFFERENCE algebraic operator.

Symbols, as used in DES90, KOR91 etc.. are of course replaced by the defined equivalent, i.e.

select	$\sigma$
project	$\pi$
union	$\cup$
intersection	$\cap$
difference	$-$
join	$\bowtie$
product	$\times$

### 3.4.7 User support

The user of the system cannot be expected to be immediately fluent in its use. A number of features should provide functionality that aids them:

**On-line help** This would include on-line help which the user should be able to take advantage of as necessary. The help facility should be accessible through the command line interface with the appropriate command(s). Each feature of the system, from executing queries, to shutting down the system, should be detailed. The system does not necessarily have to be context sensitive.

**Log file** The system will perform a number of operations behind the scenes, which the user is not necessarily aware of. A log file would record serious occurrences, and enable not so serious occurrences to also be recorded. The user may then view the file to determine what occurred to cause a particular problem.

**Debug** The creation of query expressions is not always likely to be correct first time. A feature which displays to the user the means by which the given expressions are executed would enable a solution to be found to a given query problem quicker than might otherwise occur. This feature is best incorporated with the log file functionality described above.

**Errors** The system cannot be expected to cope with every possible eventuality that occurs. As a result, errors should be reported to the user when an eventuality that cannot be dealt with occurs. A meaningful error report should occur, rather than the displaying of a number.

**Redirection** The output to the screen should be recordable. As such the concept of standard output<sup>13</sup> should be used throughout the program, such that the user can redirect output to a specified file. In addition to this, and to

---

<sup>13</sup>As it applies to MS-DOS, rather than the extensive UNIX approach.

enable the user to interactively enter queries<sup>14</sup>, an automatic log of the system events should occur into a pre-defined file.

### 3.4.8 Conclusions

Given a BNF definition of a language, any parsing program based on the definition should, given the same input, produce the same output as another.

The two approaches outlined for parsing expressions are appropriate for different environments. The first, two stage approach, enables an optimisation step to be incorporated. This feature is not necessary in a system of LEAP's scale and size, although in the long term it would be more dynamic. In the meantime the second, one stage approach is appropriate. It is simpler and just as powerful as far as the user is concerned. Its main disadvantage is its inability to optimise expressions. However this is not strictly necessary in the bounds of the project. This second approach should therefore be implemented.

A number of user aids are discussed. A sizeable subset of these features should be provided to aid the user of the system.

## 3.5 The enhancement framework

### 3.5.1 Introduction

It is beyond the means of this project to design and implement features found in larger relational database management systems. The project should however, be in a position to enable the incorporation of such features if required.

This does not stretch *ad infinitum* to current research/development concepts such as object orientated relational database management systems, but within reason to the established facilities provided by RDBMS such as indexing, hashing, data dictionaries, optimisation etc.

The structure of the LEAP system should be such that enhancements can be incorporated in as seamless a way as possible. This is achieved in all projects through the process of abstraction, and modularity within the coding.

### 3.5.2 Structure

The key to abstraction, and thereby extendibility/modification is modularisation. Each module of code should be self contained, and abstract upon the 'layer' of modules

---

<sup>14</sup>It is not a straightforward process to interactively enter data into a program, whilst recording the standard output. The standard output is usually only sent to one location, and if this is a file and not the screen, the user does not know when to enter data, or if it was entered correctly.

beneath it. Through abstraction, modules can be rewritten and replaced as necessary providing they provide at least the same functionality<sup>15</sup>.

Further modules may be added as functionality is increased, and incorporated within the higher level of abstraction. Data structures should be defined as necessary, and these data structures available to all modules that require them. Modules should only provide the functions that are necessary externally, and should 'hide' the functions that are utilised internally from external modules.

The result should be of building blocks that can be placed together appropriately, to form a structure that provides a service which is utilised by a higher level block. At the top of the block would sit the user interface, which will pass commands through the appropriate modules. The majority of the 'work' will be performed by the lower level operations.

### 3.5.3 Efforts

Effort is required in order to produce an enhancement framework. This document is part of this effort. A (hopefully) clear design which may be read, and enable the reader to gain an understanding for the structure and philosophy throughout LEAP.

In addition to this, LEAP should be written with a mind to enhancement. The code that results from the design should mirror the design, and should explain the approach taken through use of commenting.

Extensive documentation (appendix A) provides a detailed description of the expected input, output and results of the form:

<b>Requires</b>	Specific requirements that the routine has.
<b>Results</b>	Side-effects within the system, for example memory allocation/File closure.
<b>Returns</b>	The return result of the function.
<b>Functionality</b>	A detailed, yet high-level description of the routine, detailing its operation. It should act as an introduction to the routine, and commenting contained within the appropriate code explain its operation in more depth.

### 3.5.4 Proposed examples

The examples that are perhaps central to relational database management systems *beyond* the criteria specified by COD79 are improved access techniques to the relational constructs specified.

---

<sup>15</sup>Additional functionality should not alter the specified functionality of existing routines, as higher level routines will 'rely' on certain results.

These techniques include indices, hashing, and caching/buffering. The project will demonstrate the ease by which a new module may be inserted into the system and incorporated within the core system.

The provision of indices and a hashing mechanism, which can be incorporated into the relation operations discussed earlier (section 3.3) are the most appropriate. The design and implementation of these extensions is discussed within section 4 (Implementation issues), and appendix A (Technical documentation).

### 3.5.5 Use of Methods

Given that a strong enhancement framework has been provided, how would it be used to provide a new feature? There are two means by which the framework for which the framework can be used

- The addition of new functionality
- The modification of existing functionality

#### 3.5.5.1 *The addition of new functionality*

New functionality is best implemented by providing an entirely new module of routines that draws in necessary functions and data structures from *lower level* operations.

The new structures and operations will not automatically be used by higher level modules, therefore they will require modification in order to make the best possible use of the new features. For example, an indexing routine will possibly entail the rewriting of some of the relational operators to make them more efficient<sup>16</sup>.

#### 3.5.5.2 *The modification of existing functionality*

It is certainly possible that the implementation of a core feature of the LEAP system is later regarded as inefficient, inappropriate or poorly constructed. No author can safely and honestly say that their system is infallible, and beyond reproach! As such the framework should facilitate the replacement of some part of the overall functionality.

By rewriting the operations with the same requirements, and same results, but more efficiently internally, higher level operations should not require any modification. In order to facilitate this it is clear that there should be abstraction within the coding.

The new routines could possibly completely replace the routines within a given module, or there could be an alternative operation which is called given a certain set of circumstances within the required data/parameters. Essentially, what goes on the 'black box' of lower level modules does not matter, *as long as the 'expected' result is*

---

<sup>16</sup>Assuming, of course, that the indices are implemented efficiently, and overall their use is appropriate!

*achieved*. It could be a completely new ‘black box’, or a ‘black box’ with a revamped internal structure.

It should not be necessary to restructure the higher level operations. However, it cannot be completely ruled out, in which case the higher level operations which directly use the new routines will require modification.

### **3.5.6 Conclusions**

The enhancement framework should successfully provide a means by which LEAP can be extended and modified beyond its original design criteria, by later developers. This includes a framework for development, and supporting documentation. The implementation should provide appropriate examples of such enhancements.

### **3.6 Design summary**

The design of LEAP has been discussed at some length. It has analysed the requirements of a database management system that supports the relational theory. A series of design decisions have been made with regards to the expected way in which these requirements will be implemented.

The implementation of the design is discussed in the following section. The success of the implementation with regards to the design is discussed in some detail in the conclusion.

## 4. Implementation issues

### 4.1 Introduction

The implementation of the database management system should adhere to the specified design. The design is deliberately flexible to enable implementation issues to be appropriately formative in the structure and end result of the system.

This section will discuss the wider issues surrounding the implementation of the database system, and the issues regarding the actual implementation. It will not discuss in great detail the implementation of specific routines, as the technical documentation appendix (Appendix A) contains this.

### 4.2 The implementation environment

#### 4.2.1 Introduction

The environment in which the design is to be implemented depends on a number of factors:

- available resources
- knowledge of these resources
- prior experience
- personal preference

It is clear that there can be no 'correct' environment choice.

#### 4.2.2 Available resources

The resources available for the project are limited. The computing and mathematical sciences department at Oxford Brookes University has available a heterogeneous UNIX network, with Sun and Digital machines running different implementations of UNIX. The computer services department has a large PC network, with a few UNIX machines available for students to use with appropriate permission. Personally, a PC is available.

#### 4.2.3 Knowledge of resources

Through work experience taken as part of the computing degree course, UNIX has become a familiar environment - but not from a development perspective, more from a users perspective. However, the use of UNIX for such a task does not necessitate an in-depth understanding of the workings of the operating system. However, an understanding of the system is bound to be beneficial.

The PC environment is increasingly becoming the 'standard' desktop system. Through work and personal experience, an understanding of the system and a broad but limited knowledge of the operating environment (usually MS-DOS with MS-Windows) is available.

#### **4.2.4 Prior experience**

Through work experience, knowledge of UNIX and PC's has been extended. It would be only prudent to take advantage of this prior experience when deciding upon the environment to use. Placed in the balance, knowledge of PC's greatly outweighs the knowledge of UNIX machines.

#### **4.2.5 Personal preference**

Because of the availability of a PC personally, the environment offered by it is preferable. The system is available at will, whereas college computers are used by other students undertaking project work and coursework. As deadlines approach, the resources normally freely available are scarce.

#### **4.2.6 Conclusion**

The greater knowledge of PC's, and the free availability of a machine ensures that this environment is appropriate. However, it is not necessarily the ideal. In an ideal situation, the environment would be the most appropriate for the system to be developed, with a mind to the eventual users.

Who are the eventual users in this scenario? This is difficult as a project of this ilk does not have a clear cut idealised 'user' in the same way a system programmers tool might. If a 'user' has to be identified, it would be a student (at whatever level) of database theory, wishing to experiment with database management systems, and algebraic operators.

As PC's are prevalent in most organisations, the environment they offer is appropriate for such a wide audience. UNIX is used at many academic institutions, but it is a multi-user system with many constraints. It also appears in many 'flavours' and versions. Which would be appropriate? Standards do exist, but how well supported are the standards in the different 'flavours'?

It is therefore clear that a PC is appropriate for personal reasons, and also for the wide audience identified.

### **4.3 Implementation language**

#### **4.3.1 Introduction**

Given that a PC environment is to be utilised, the next step is the decision as to the implementation language to use. The PC environment offers an ever increasing range

of programming languages and paradigms which offer a great deal of flexibility and functionality to the developer

The language that will be used is an important decision which will affect the final outcome of the system. Two factors affect the choice: Available, and knowledge of these languages.

### **4.3.2 Available resources**

What resources are available? Given the large number of packages on offer at the college computer system, including a wide range of programming language, the choice is not really limited.

However, implementation will primarily take place at home, where licenses must be obtained. The packages for development with existing licenses include Basic, 'C' and Pascal.

### **4.3.3 Knowledge of the languages**

Knowledge of the languages is an important criterion. It is not desirable to have the overhead of learning a language in addition to the implementation of a system.

This applies to 'C'. At the outset of the project, knowledge of the language was patchy. Whilst knowledge of 'C' has extended since the outset, the decision as to a language had to be made early.

Basic is known, but not extensively. Basic has a number of incarnations, from Spectrum Basic, to Visual Basic. It is a simple language geared for beginner programmers, more for prototyping (especially in Visual Basic) than serious system development. Basic does stand for "Beginners All Symbolic Instruction Code" after all. It is clearly not suitable for the documented system design.

Pascal on the other hand, is known more extensively. It is a well known general purpose language that offers a great deal of functionality. The Borland implementation of the language offers a large number of extensions to the Pascal standard, bringing it features such as modularisation, object-orientation, and a wealth of development tools.

### **4.3.4 Conclusion**

The Borland Pascal implementation language is the obvious choice. Licenses exist and it is well known both personally and to others, and is flexible.

## **4.4 Implementation approach**

### **4.4.1 Introduction**

The approach that is taken during implementation towards the system design documented earlier is clearly important to the success of the system. Of most

importance is the vague design criterion “A well-defined framework for future enhancement”. The other criterion are specific goals to be achieved, visible within the code as system functionality.

## **4.5 Implementation issues**

### **4.5.1 Introduction**

The implementation of the system brings a number of issues, both large and small in importance. This section will deal with these issues, explaining the decisions that have been made.

### **4.5.2 Relation naming**

The creation of relations occurs within every valid relational expression. The majority of relations, certainly within a nested expression, will be temporary relations that will not be of concern to the user. As such, the user will not have an interest in the naming of such relations.

Yet, a relation must have a name. This is specified in the design. It must be possible to refer to relations by some convention, in a form understandable to the user.

What names should be used? It is clear that the number of relations that will be created, either permanent or temporary, is not known at any stage. Effectively this number is random to the system. It is therefore not possible to have some database of relation names that can be used one by one. There is no certainty that all the relations will be temporary, which would enable this list of relation names to be reused when exhausted. They could all be permanent - and no two relations may have the same name.

The answer is to enable the system to generate relation names as necessary using some algorithm. This algorithm should ensure to a satisfactory degree that the same name will not be generated twice within the same session.

Some random name generator is required, which creates relation names. As the relation name is not of concern to the user, it does not matter that the relation name is meaningless.

Borland Pascal provides a random number generator that must be initialised with `randomize` before use, and `random` to generate a random number between zero and the specified number minus one. These functions can therefore be used to create random names by generating numbers between zero and 25, and converting the number to the appropriate ASCII characters representing the letters of the alphabet. A sequence of calls to the `random` function can generate a unique name for the relation.

Yet how random is the name? And how certain is it that the same name will not be generated twice?

The random number generator is suitably random for it is dependant upon the computer clock which is initialised at start-up, which will ensure that it is at a suitably random position to give at least a pseudo-random number. It is not truly random, but more than adequate for the purposes of the problem.

The name is therefore adequately random. Without considering the ordering factor of characters within the name, the probability of the same name appearing twice is 1 in  $1.12 \times 10^8$  when 5 characters are used. As the maximum name of a relation is the same as the maximum name of a MS-DOS filename, i.e. 8 characters, this gives the probability of the same name appearing twice as:  $2.08 \times 10^{11}$ . For the current system, this is more than adequate.

Yet there is still the (distant) possibility of the same name appearing twice. Clearly this must be considered. A check is made when creating any relation for the same name within the database. If the relation is found, an error is reported, and all dependant operations abort. The likelihood of this occurring because of a random name clash is not in the least a significant possibility.

### **4.5.3 Command processing**

#### *4.5.3.1 Introduction*

Command processing is an important component of the system as a whole. It will dictate how the underlying functions are utilised, and to what extent the user can interact with the data. The approach taken at this level is not however, as important in terms of the enhancement framework as the implementation of the relational structures and operators.

This section will discuss the possibilities that exist, and the route that has been taken, further enhancing the brief design decision.

#### *4.5.3.2 Options*

There exist two main options for the command processor element of the system. The first is an essentially two stage processor which processes an expression, building an internal representation of the query, followed by the execution of the query. The second is a one stage processor, which processes and executes the query at the same time.

Both approaches, similar yet distinct, perform the same function - to provide the user with the services offered by the lower level operations.

The first approach is taken by all conventional commercial interfaces, i.e. Oracle's SQL\*Plus interface. A prompt allows the user to enter queries, and the query is parsed, and executed by the Oracle RDBMS.

There is a clear reason for the two stage approach. The building of an internal representation of the query allows query processing to occur. It is certainly the case that queries are rarely entered in their most optimal form, especially when the concept

of views is introduced. By processing the result of the parsing, the query can be modified such that it is in a more optimal form. An optimal form of query will ensure that the most efficient use of the RDBMS is made. The optimised query is then passed to the RDBMS and executed.

The second approach is far simpler, yet less flexible. Essentially recursive in nature<sup>17</sup>, Each individual expression within the query is processed separately, and the result passed back to a 'higher' expression.

The benefit of this approach is a far simpler implementation, yet it is clearly not going to reach the efficiency of the two stage approach. No pre-processing of the query occurs, and it is executed exactly as the user requires.

The two approaches discussed are both feasible, yet it is not possible within the scope of the project to implement both. The merits of the first would certainly outweigh those of the second in a commercial system, where efficient use of resources is required. However, in a system such as this there is little room for commercial considerations.

The first, two-stage approach is clearly more desirable. However, it is an inherently complicated approach, that is a project in itself. For these reasons alone, it is not a reasonable approach to take. The second, one-stage approach is not as desirable, but it does offer something the first approach cannot offer: flexibility. The 'audience' of the system was earlier sketched as academic. It would be desirable for this audience if the ability to attempt different query structures were enabled. The first approach would attempt to ensure the most efficient structure automatically, the second does not - which would demonstrate to the audience how queries can be directly improved by following optimisation techniques outlined in most database system texts (See the Bibliography)

#### 4.5.3.3 Approach

Given the second approach, whilst not as technically meritable as the first, is to be implemented, a structure is required. Clearly the approach is recursive in nature, and that a recursive implementation is the simplest. An iterative implementation could very well be implemented if desired, but this would further complicate the matter.

An expression within the system is to take the form of the relational algebra operators, operating on one or two relations, which may themselves be further expressions.

For the following discussion, the following relation will be used:  
`names ( name , address , tel.no )`.

For example, the projection of the name and address attributes from the names relations would take the form (see section 3.4.3 for a BNF definition)

---

<sup>17</sup>Although it is perfectly possible to implement an iterative approach, to conceive of the processing as recursive is simpler.

```
project((names)(name, address))
```

If only the name and address of persons living in Oxford were to be projected, the expression would take the form;

```
project((select((names)(address='oxford')))(name, address))
```

The operation cannot complete without selecting the residents of Oxford prior to the projection of the name and address attributes.

The first approach detailed would assess whether it was possible to project prior to a select (it is), and whether it would result in a more efficient means of executing the query. An advanced implementation of this approach would also determine whether significant time should be spent determining the most efficient approach, which can in itself lead to longer than necessary execution times.

The approach decided upon would execute each expression in turn as encountered, executing sub-expressions as necessary.

In the above example, the project operator's expression would be assessed, and the relation on which it operates examined. As the 'relation' is in fact a further expression (a select), this would need to be assessed and executed. This might entail further expression evaluation. Once the expression has been resolved, the resulting relation has to be passed 'up' to the project operator. The project operator can then proceed to project the name and address attributes.

Therefore, the core of the approach will revolve around a routine which will determine whether an expression contains further expressions, or a simple relation, and returns the resulting relation. Recursive in nature, it will call itself again if there is a sub expression. The routine will call the algebraic operators in order to generate the resulting relation.

The algorithm for the routine follows:

1. Retrieve "next" expression/relation
1. If result is a relation, return the relation and exit
1. If the result is an expression execute steps 4 then 5.
1. Evaluate each 'relation' within the expression by calling the routine at step 1.
1. Execute the algebraic operator with the returned relations. Return the resulting relation and exit.

This routine would only be necessary when the command entered required that a query be executed, some pre-processing is clearly necessary to determine whether the command entered is an algebraic expression, or simply a 'maintenance' command.

#### 4.5.3.4 Implementation

The implementation follows the given algorithm closely. Expressions are determined through extensive use of brackets. A relation/expression is required to be surrounded by a set of brackets. Each set of brackets where a relation is expected is cut out of the remaining expression, and the contents recursively passed to the routine. The routine repeats this procedure until it has resolved the expression/relation, at which point a relation is returned.

The returned relation is passed back to the calling routine, which may repeat the process for the next relation (if it is a binary operator). The algebraic operator is then executed, and the result returned, which is returned back to the last expression evaluation, or possibly when the routine has completed the evaluation of the expression. The result after the routine has completed is the result of the entire expression, which may or may not be valid, i.e. if an error occurred.

A severe restriction in this approach is the inability to analyse the expression prior to evaluation. This evaluation is not necessarily purely for optimisation purposes, rather it may also occur to check the validity of the expression. For example, the expression may not contain a valid relation name, or may contain incorrect bracketing. Without such pre-processing, the routine must deal with an error when it is encountered. If this at the end of an expression, there may be much wasted time.

A means around this is the provision of some form of debug output, which enables the user to view the progress of the expression evaluator (and in a sense visualise the parse tree given in **Figure 3-9**, on page 28). Through this, the expression can be dissected and modified to achieve the correct result.

#### 4.5.3.5 Conclusions

The decided approach is not the most efficient, but it is the most appropriate given the scale of the project, and the anticipated audience. It does not necessarily encompass the philosophy of an enhancable system structure, whilst the first approach clearly does. Yet concessions must be made in a project of this scale.

The lack of checks prior to the execution of the expression is restrictive for on the fly, nested expressions, but it is not expected that the system will have to cope with particularly complex expressions.

### 4.5.4 Data structures

#### 4.5.4.1 Introduction

Data structures are an integral and important part of any system, as highlighted in the title of WIR76, “*Algorithms + Data Structures = Programs*”. This section will discuss the significant data structures that are a part of the system.

The design documentation detailed the high level structures. This section will detail the low level considerations. There are a number of such structures:

- The master relation structure
- The tuple structure
- Stacks
- Indices/Hashing

Each will be dealt with in turn.

#### 4.5.4.2 Relation record and data structure

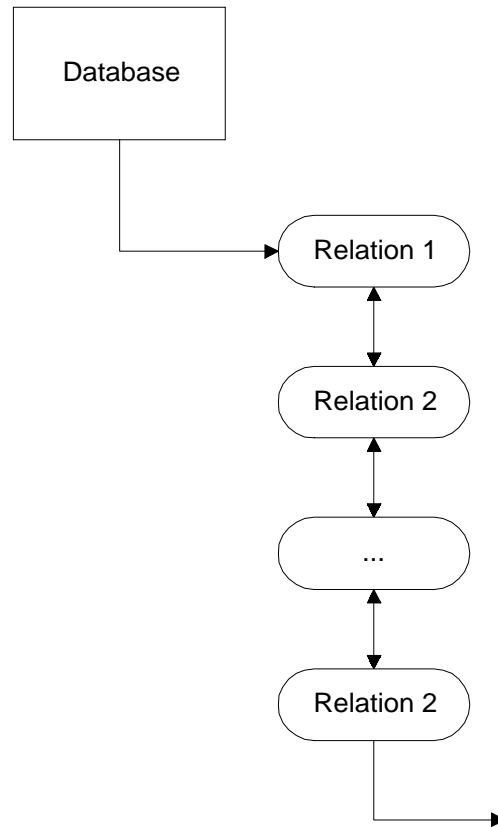
The master relation structure links all relations that exist within the system. This ensures that all relations can be located quickly and easily.

The structure is referenced in the master database structure, which contains a pointer to the first element of the list. Each node within the list contains two pointers to the next and previous nodes. The last elements 'next' element is `nil`, and the first elements 'previous' element is also `nil`.

Each relation node is defined as a record containing reference information to the physical secondary storage record, and internal definition information, with structural information:

Record Field Name	Data Type	Description
<b>name</b>	string	Contains the name of the relation
<b>filepath</b>	string	Contains the path to the relation "body", containing the tuples.
<b>filename</b>	string	Contains the name of the relation "body" file.
<b>fieldname</b>	string	Contains the name of the relation "header" file
<b>fileptr</b>	text	File descriptor for the relation body file.
<b>fieldptr</b>	text	File descriptor for the relation header file.
<b>storage</b>	(record)	Record structure containing the "next" and "previous" pointers.
<b>nofields</b>	word	Contains the number of attributes that exist within the relation.
<b>Temporary</b>	Boolean	Boolean flag that indicates the temporary status of the relation.

This structure is graphically represented as in Figure . Each node within the diagram can be visualised as the above table.



**Figure 4-1 - Relation Data Structure**

A number of operations are required on the relation structure, and these are documented in section 3.2.3.2.

#### 4.5.4.3 *The tuple structure*

The tuple structure is clearly important to the operation of the algebraic operators. It is therefore necessary to implement it appropriately. The design discussed the issue of representing the tuple, and an array structure was specified as the appropriate method. However, an array can be defined in a number of ways: As a local/global variable, or as a field within a dynamic structure.

The first option would clearly be very limited, and would tie the number of tuples that may exist to the implementation code, whereas the latter option is more dynamic and enables an unknown number of tuples to be used and referenced.

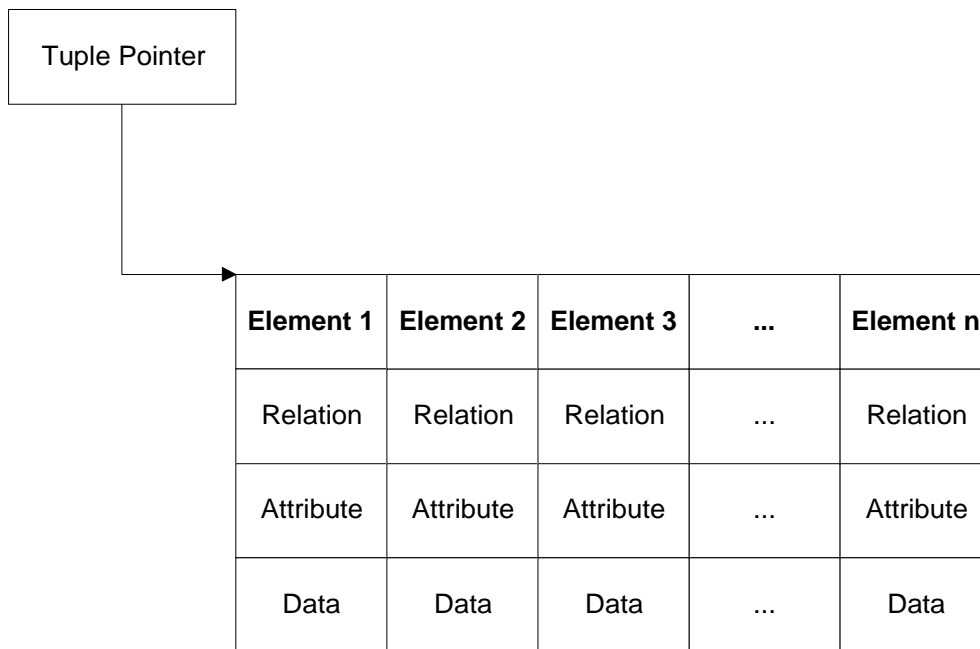
A tuple node will therefore consist of an array, containing attribute and value information, in addition to information deemed necessary (i.e. the relation to which the attribute/value pair originates).

A tuple record may therefore be visualised as an array of the following records:

Record Field Name	Data Type	Description
<b>relation</b>	relation pointer	Contains a pointer to a relation from which attribute/value pair originate.
<b>field</b>	attribute pointer	Contains a pointer to the attribute from which attribute/value pair originate
<b>data</b>	string	Contains the data for the attribute/value pair.

The tuple data type is therefore a pointer to a record containing an array of the above structures. The array must be of a given degree, defined at compile time. The size must be as large as the anticipated maximum number of attributes.

In the long term, this approach is clearly inflexible. It enables a variable number of tuples to be stored internally within the system at any one time, but limits the degree of relations. The best approach, although not as cohesive with the implementation language structures, is some form of dynamic array. This would enable tuples to be limited only by available physical memory.



**Figure 4-2 - Tuple Data Structure**

Each operation on the physical representation of the tuple will create such an internal structure as represented in Figure 4-2. Because the tuple structure has a fixed size array, not all of the elements will be populated. Pointers would therefore be set to `nil` if no data is present. The relation and attribute fields within the diagram would contain pointers to the appropriate internal structures.

The operations that are necessary to act upon the structure are documented in section 3.2.5. In addition to these, a number of other routines will be necessary to build the tuple:

<code>Tuple_concat</code>	Concatenates two specified tuples. This operation is used by the join and product operations.
<code>Tuple_to_string</code>	Converts a given tuple to a string equivalent. This would be necessary in routines that require some hashing operation.

The `build_tuple` routine specified earlier in section 3.2.5 would be the most useful, for after an algebraic operator has created the appropriate relation, a call to this function will build an 'empty' tuple ready for data to be populated. The routines for reading tuples from disk should create the tuples automatically as necessary.

There is no link between the relation and the tuples loaded in memory simply because there is no need for such. A relation is produced within an algebraic operator, and the operator will store relations if and as necessary. When the operator has completed, all tuples are appropriately disposed of, as there is no further need for them.

#### 4.5.4.4 *Stacks*

A number of stack structures are necessary<sup>18</sup> throughout the program. The discussion will be generic across the stack implementation.

Stacks are used with the relation structure (to store temporary relations that are created by the relational operators, and dispose of them together<sup>19</sup>), and also within the index module to enable an iterative search of the B-Tree structure to take place.

A stack requires two separate record types. The header structure contains reference information to the actual data structure. The element structure contains information about each element in the structure, and a pointer to the next node in the structure.

---

<sup>18</sup>Pascal, including Borland's implementation (without using object orientated methods) does not enable a generic data structure to be created, and applied to different data types. This feature is available in Ada, and other such languages which have identified this shortfall. Object Orientated methods do enable this, but have not been used because of the additional complexity that would be introduced within the program.

<sup>19</sup>A queue structure could also be used, but a stack is slightly simpler, and previous code could be reused and modified.

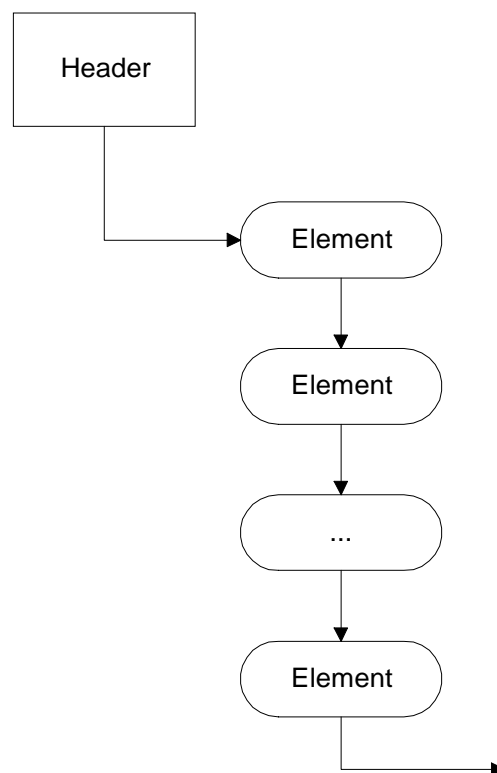
Header Structure:

Record field name	Data Type	Description
<b>head_node</b>	node pointer	Contains a pointer to the first node in the stack
<b>no_items</b>	word	Contains a record of the number of elements in the structure.

Element Structure:

Record field name	Data Type	Description
<b>Data</b>	[Ptr]	Usually a pointer to some structure. It contains the data held in the node
<b>next</b>	node pointer	Contains a pointer to the next element in the stack.

Graphically, the stack may be visualised as in Figure 4-3.



**Figure 4-3 - Graphical representation of a generic stack**

A number of functions are necessary to act upon a stack (based on STU89):

**Push** Adds a specific element to the stack structure as its most recently arrived element

<b>Pop</b>	The most recently arrived element of the stack structure is removed and returned,
<b>Empty</b>	If the stack structure has no elements, then empty is TRUE, otherwise empty is FALSE.
<b>Size</b>	Size contains the number of elements in the stack structure.
<b>Flush</b>	All elements are removed from the stack structure.

#### 4.5.4.5 Hashing

The idea of hashing is to store information according to some key, and to use the same key (generated at a later date) to retrieve information at a later date. Hashing uses a *hash table* which contains the different keys (and the appropriate information), which are distributed evenly through use of a *hash function*. The hash function returns an index to the hash table. The ideal hash function would generate a unique index for every possible index. This is clearly not possible with constraints on hash table size. Therefore collisions occur, with two records needing to be stored in the same location. Different methods exist for generate the hash key, and for resolving collision. (KRU84).

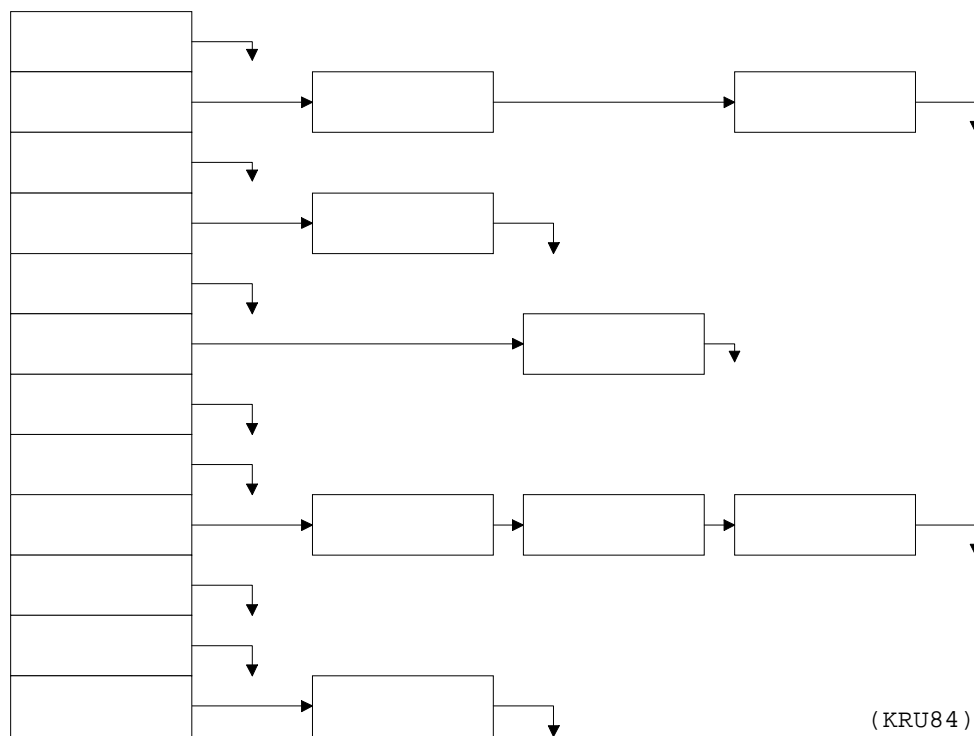
For the hash key, implementation uses the folding function, which partitions the key into several parts. The parts are then combined in a convenient way to obtain the index for the hash table. This was used because it is a relatively simple method, yet is more than adequate for the needs. Other techniques are either insufficiently flexible, or overly complicated.

The collision resolution approach is to use chaining. Each record within the hash table contains a linked list of information. When a collision occurs, the new key is added to the linked list if appropriate. This is advantageous because it ensures a dynamic hash table, whereas 'conventional' approaches<sup>20</sup> are limited to the defined array size. Deletion of keys is also simpler, involving the simple removal of a node from a linked list.

The hash table can be best visualised as in Figure 4-4.

---

<sup>20</sup>Such as linear probing; clustering; incrementing etc.



**Figure 4-4 - Graphical representation of a hash table**

Hashing is essentially used as a form of temporary indexing. A hash key is generated and the data inserted into the hash table. Subsequent keys are generated and added. A test can be made for the existence of a key in the table. This is of particular use with regard to preventing duplicate tuples existing within a relation.

The `union` operator for example combines the data from two separate union compatible relations. No duplicates should exist in the resulting relation. By populating a hash table for the output relation with a key containing the information within the tuple (for example, primary key information), a check can then be made *prior* to the writing of the tuple to the output relation. If a collision occurs, then the tuple is already present in the relation and should not be written.

#### 4.5.4.6 Indexing

The issue of indexing is not a part of COD79's definition of a component of a relational database management system. Indexing is however a feature of all commercial RDBMS. This is entirely because the provision of indexes can significantly improve the speed of some relational operators, and the operation of the system as a whole.

Indexing is therefore treated as an extension that should be considered, but not as part of the core functionality of the system.

The implementation of indexes usually uses trees, specifically balanced trees, or B-Tree's. A number of incarnations of B-Trees exist, such as the B\*-Tree, and B<sup>+</sup>-Tree.

The B-Tree is considered here for its relative simplicity and appropriateness<sup>21</sup>. Balanced tree's ensure that the height of the tree is as small as possible. This is accomplished by not permitting empty sub-trees to appear above the leaves, and ensuring that all leaves are on the same level. Finally, every node is assured a minimum number of children. Usually all nodes (except leaf nodes) have at least half as many children as the maximum possible.

The nodes within a B-Tree will contain an index of some definition, but not the entire tuple it represents. In addition to this information, the index will contain a reference to the location within the relation file where the entire tuple may be located. This ensures that the B-Tree can be searched to locate a particular node (or traversed to locate all nodes in a particular ordering), and the entire tuple read from the secondary storage medium.

The reason a B-Tree is used is because it is relatively fast to locate given tuples because of the minimum possible height of the tree. Without indexing, the entire relation would have to be searched to find a given tuple. This would require, on average, accesses to half the records in the relation (DES90). Tree structures are only appropriate (linear structures would not, on average, reduce the search time), and simpler tree structures (i.e. binary search trees) do not have a balanced structure, and therefore would not be particularly advantageous.

It is conceivable that a large number of indices may exist for a given relation, and that therefore a large number of indices may exist for the entire database. Some structure is necessary for the ordered storage of the indices *themselves*. It would be prudent to draw upon past decisions. The storage of relations within the database structure itself is the most appropriate. This used a linked list structure because other approaches were not appropriate in the context of a project of this size.

The same clearly applies here. To use a more efficient search structure such as a binary search tree would be certainly more efficient in terms of searching, but would add to the conceptual complexity of the system. The system has deliberately been constructed to be extendible and modifiable. Structures may be switched in and out as necessary. Given this, the linked list is appropriate at this stage.

Therefore a linked list structure will contain a pointer to the index structures themselves. Whenever a new index is created, it will be added to the index structure. The ordering of the index structure will be different from that of the relation structure, for a number of indices may exist for a given relation. Some use must therefore be made of the keys by which the index is created, such that the order of the index structure is maintained.

The index itself has been specified to be that of a B-Tree. The implementation of an efficient index data structure is a complex task, for the conceptual complexity of a B-Tree is several degrees more than that of a singular linked list. Therefore published

---

<sup>21</sup>The other B-Tree techniques offer significant advantages that are generally specific to given requirements. The standard B-Tree technique is more than adequate.

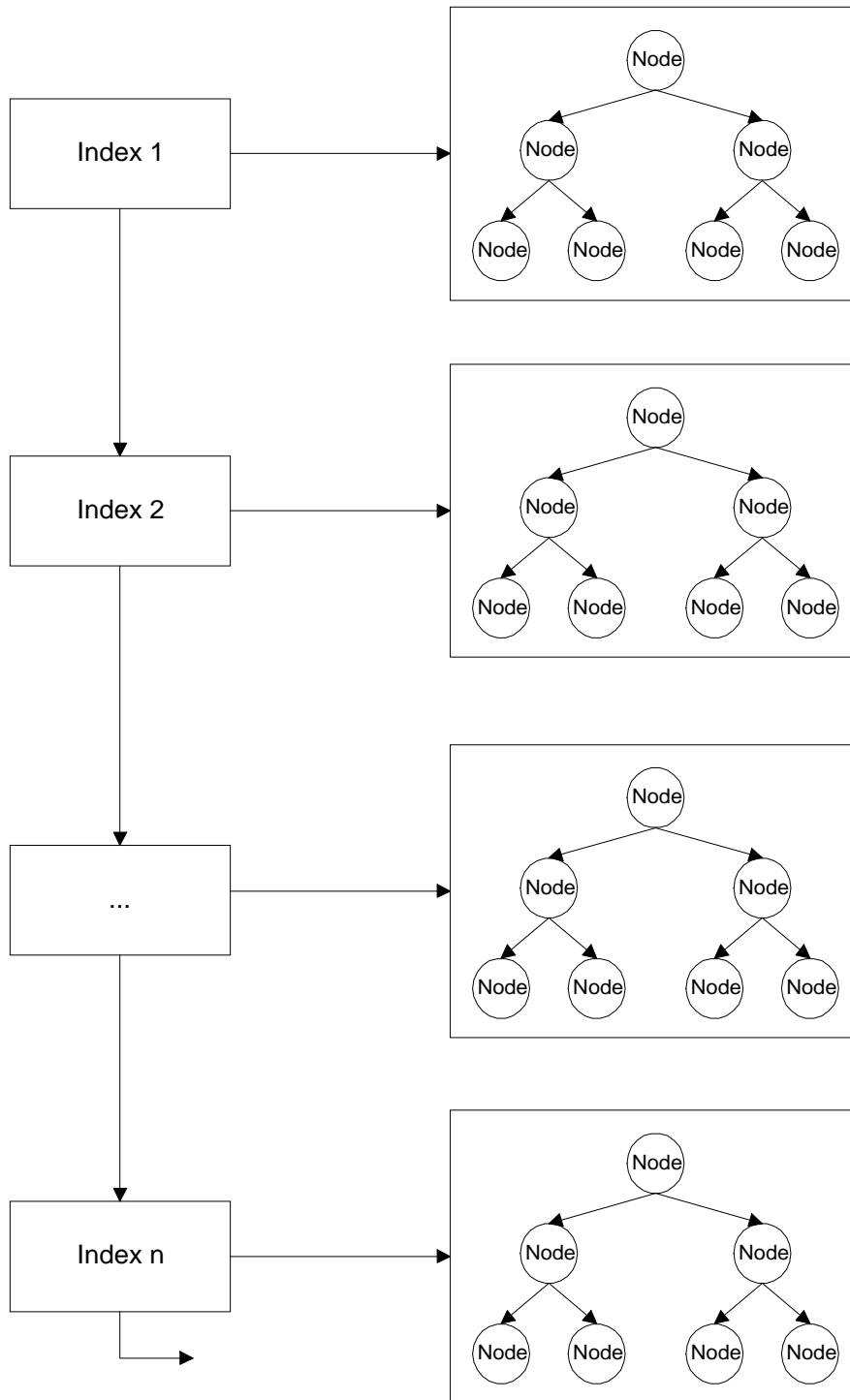
examples have been consulted to both save time, and reduce effort. The example published in KRU84 has been used. This example uses recursion to maintain the B-Tree index, but an iterative approach to search and locate information. Only the routines for tree maintenance have been used.

The primary use of the index will be to retrieve a relation in an ordered manner, for use within a single pass join algorithm. This will require the ordered traversal of the B-Tree. This is not possible using a recursive approach. A recursive approach would have to call a routine from within the recursion, and in order to be dynamic require a procedure to be specified within the parameter of the search routine.

Whilst procedures can indeed be passed as parameters in standard Pascal, it is not a particularly elegant method, nor is it particularly efficient with regards to memory. An iterative approach is far more appropriate.

The iterative approach requires that a stack of elements to be maintained, which is modified as the search algorithm moves around the B-Tree. A series of calls to a findnext class of operator will locate the next node and return it, using and modifying the stack. The findnext operator will continue to traverse the tree until all nodes have been processed, before terminating.

Clearly whenever a tuple is added to a relation, the indices will have to be updated to ensure that they are valid, and either rebuilt or retrieved from some permanent store between runs. Rebuilding the index is not particularly efficient, therefore the indices should be stored as necessary for retrieval and a speedier rebuild.



**Figure 4-5 - Index structures**

## 4.5.5 File storage

### 4.5.5.1 Introduction

The storage of data on the secondary storage medium is clearly important for the continued usage of the system. The structure of files has been alluded to in the previous sections, but specific considerations have not been dealt with.

This section will cover the issues relating to storing the data within relations.

#### 4.5.5.2 Data

The data that must be stored is specified in an earlier section (section 3.2.4.1). Essentially, the header of a relation will contain:

- Attribute name
- Data type

And the body will contain:

- Data values pertaining to each attribute.

#### 4.5.5.3 File structures

The external structure of the files has been briefly discussed, with attributes/values being stored in the same ordering, thereby removing the necessity of storing cross reference information.

What ordering should be used? COD79 states that attribute order is insignificant within a relation. No benefit will be gained by ordering the attributes in any special way, i.e. alphabetically, because there is no way that the system can predict which attributes will be used over others<sup>22</sup>. As such, the simplest approach is as good as the most complex.

The simplest approach is to create attributes within relations in the order they are specified. Therefore, a create relation command specifying relation *R* contain attributes *rl,bl,xl* would create the appropriate files with attributes in that order, rather than any other.

The implementation environment is MS-DOS, using Borland Pascal. A typed file can be used within Borland Pascal, such that records are stored in a binary format for easy retrieval within Pascal. This has the advantage of speed, but does not enable the user to view the files on the secondary storage medium outside of the program. It is felt that a user may wish to inspect the relation definition (and data) outside of the bounds of the program<sup>23</sup>.

The appropriate approach is therefore to store the data in a textual format, using ASCII characters. Pascal enables this through use of a `text` file descriptor rather than

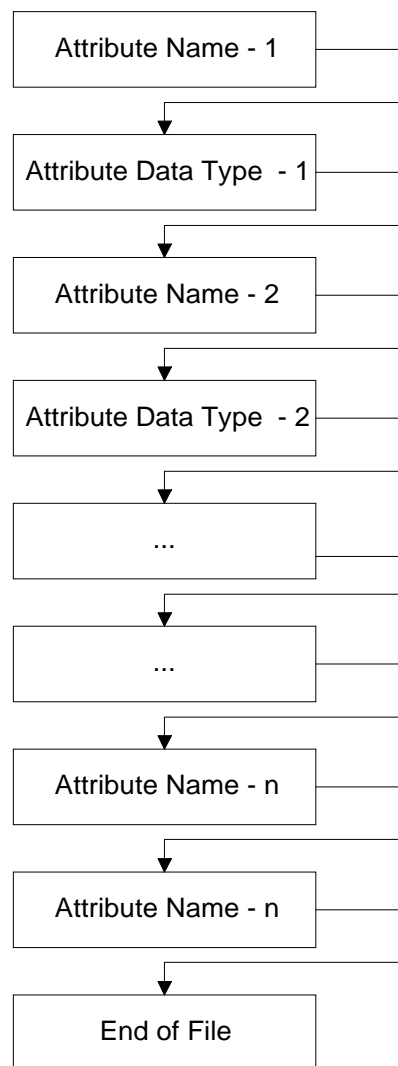
---

<sup>22</sup>If there were a way of predicting the usage, it wouldn't necessarily follow that the ordering would improve the operation, i.e. frequently used attributes placed *before* less frequently accessed attributes, as certain positions within a file can be located just as easily as others in most file accesses.

<sup>23</sup>It may be necessary for the user to make changes for some lack of functionality (such as update/delete). If the functionality of LEAP were suitably enhanced, it will become less likely that the user will need/want to, and restrictions could be imposed.

a typed file. The Pascal `readln` and `writeln` procedures are used to read and write to opened files, providing they are properly set for reading/writing. The file, once closed, can then be read by other programs.

Given that the attributes are not ordered other than by the user when creating a relation, the file may therefore contain an attribute name, followed by the attribute data type. The data can be written, one item per line (two lines in total per attribute). Thus:



**Figure 4-6 - Attribute/Header file structure**

The structure of the body is more complicated. It again follows that a text file should be used, rather than a typed file. Because the system does not enable the data held within a relation to be changed (no support for the insert-update rules), the user will still possibly wish to change tuple attribute-values. It would also be prudent to store each tuple on separate lines. The tuple should be ordered as per the design specification (section 3.2.5), with attribute-values ordered in direct correspondence with the header structure order.

There are two main approaches:

- Each tuple attribute-value is separated from the next attribute-value by a special character
- Each tuple attribute-value is separated from the next by a specific amount of whitespace<sup>24</sup>

Both are possible, but the whitespace approach is preferred because it is possible that the user may wish to store information containing the defined special character. In order to process the tuple, the tuple is first read from the secondary storage medium. As each attribute-value has a maximum size, the trailing whitespace can be removed. The remaining data will be the previously stored data. The process is repeated for the entire tuple's data, which is terminated by a new-line marking the end of the line. The process would be repeated, in reverse for the storage of tuple-values.

#### **4.5.6 Operational structure**

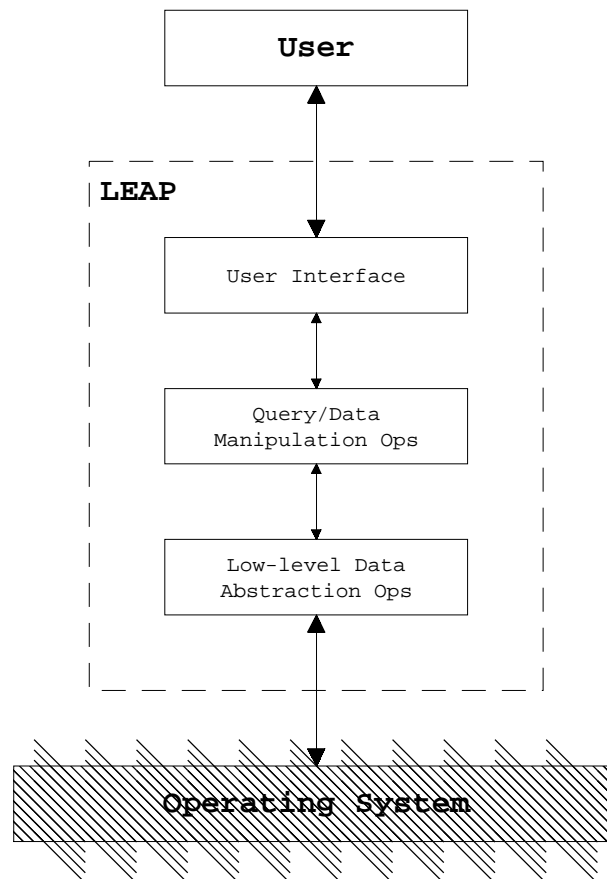
The way in which the system has been conceived and implemented as operating is discussed in this section. More details are provided in Appendix A.

The system essentially consists of three layers (graphically represented in Figure 4-7):

- A high level command line interface
- A level of data manipulation and query operations.
- A low level series of operations, which manipulate the physical and internal representation, to present the structures necessary for the relational concept.

---

<sup>24</sup>Whitespace is classed as separating characters, such as spaces, carriage returns etc. That do not alter the meaning or content in any way.



**Figure 4-7 LEAP System Abstraction**

Commands are entered by the user into the command line interface. The commands are processed, and if appropriate passed to the middle layer of operations<sup>25</sup>. There is very little direct communication between the high level and lowest level. Some communication is necessary to extract data from a particular data structure.

The user may enter a `project` query. The command line interface determines that the command is a project operation, and calls the appropriate operations within the middle layer.

The middle layer uses the low level operations to provide access to the physical representation of data. The higher level only “sees” the requirements and output of the middle layer operations. Likewise, the middle layer only “sees” the requirements and output of the lower layer operations. It is not “aware” of the higher level.

The project operation therefore breaks down the attributes to be projected from the specified relation, determines whether they exist within the relation, and extracts the appropriate data using the low level operations.

---

<sup>25</sup>The majority of commands will be passed through to the middle layer. Those that are not will be internal commands for the command line interfaces operation, such as variable setting, command history facilities etc.

The low level operations directly interface to the operating environment, i.e. the operating system, using language facilities where necessary. The low level only “sees” the “coal face”, and provides a level of abstraction more congenial to the implementation of relational operators, e.g. the concept of a tuple is provided through data structures maintained by this level.

## 4.5.7 User support

### 4.5.7.1 Introduction

The user support features discussed in the design section (section 3.4.7) are discussed with regards to implementation.

### 4.5.7.2 Redirection

The Borland Pascal environment allows the output from a `w r i t e` statement to be directed to a given file which has been duly assigned to a file descriptor, and opened for output. The standard output file is a special class of file, whose filename is an empty string, i.e. `''`.

In order to enable this to be easily implemented, the file should automatically be opened for writing. A number of procedures should then facilitate the writing of data to this file. All output then uses these procedures.

If at a later date, this feature was no longer required, the special procedures can simply write the output to the screen, rather than through standard output.

The need to record all data in a separate file can be encompassed within the routines displaying the output to standard output. In addition to writing to the standard output, they should also write, through an additional set of routines, to a specified file which is created as appropriate. The file should be written to the appropriate location, which would by definition be temporary, and therefore go to a file within the specified temporary location, usually specified by the `temp` environment variable.

### 4.5.7.3 Log file

In a similar manner to the standard output log file, a file which contains the status of the system as it progresses is to be implemented. A series of routines which write to this file should therefore be created. Each routine which needs to report an occurrence can call the appropriate routine, and the appropriate data be written.

As the file will stretch across execution sessions, it should be opened for appending, which would not erase any data previously stored. As the file could conceivably grow rather large, the user should be able to restart, or ‘flush’ the file.

#### 4.5.7.4 *Debug*

A special form of report should be possible, which records progress rather than eventuality information. This information would be prefixed with the `DEBUG` string, and appear in the log file discussed in section 4.5.7.3. This information is not always necessary, and the user should therefore be able to disable the feature as necessary.

The most appropriate use of this facility would come with the analysis of expressions. As the system commences evaluation of each expression, or sub-expression, a debug report would be written. When each expression has been evaluated (successfully or not), a debug report is also written. The user will then immediately be able to see the progress of an expression as it is evaluated, and determine where the problem lies.

#### 4.5.7.5 *On-line help*

On-line help should be accessible through execution of the help command. This would bring up one or more pages of information pertaining to the use of the system.

There are two approaches to coding such a help facility: The help page could be hard-coded as part of the program, or the help page could be a file on secondary storage which is referred to and displayed.

The first approach is clearly inflexible. A recompilation is necessary if the help page is changed in any way. It is faster than referencing a file, but there is no specific speed requirement within the system, and the difference is not particularly significant to the end user.

The second approach is far more flexible. The help facility need only read each line and display it to the standard output (through the routines discussed in section 4.5.7.2).

It is not necessary in a system of this size to have context sensitive help, such that help can be brought up on the `project` operator whilst the user is coding a query featuring a project operation. This would be beneficial, but in a command line, text based implementation (with output redirection) it is not cohesive with the overall philosophy. However, it should be possible to display portions of the help file, such that a given 'page' rather than the entire file can be viewed. The file should therefore be structured into logical 'pages', which would contain similar information.

Each page would need to be separated in some form. A single pre-defined character on a line of its own is adequate. The help routine, if given a particular page number, would therefore have to search through the file until it has encountered the appropriate number of such separators. The text between the separator, and the next such separator should then be displayed.

#### 4.5.7.6 Errors

An error encountered within the program should report a helpful error, rather than a cryptic error number and termination<sup>26</sup>. Therefore, throughout the implementation, the implementation environments error handling features should be disabled, and the program itself take over responsibility. Errors are then raised by the implementation environment, but set a variable rather than halt the execution.

Errors should then be handled 'politely'. This may still result in system shutdown (some errors are classed 'fatal'), but a more helpful and informative error should be displayed.

To display a helpful error message requires that a set of pre-defined errors should be specified within the program. Such errors should have a defined constant value, such that the same error number is displayed for the same class of error, i.e. 'File not found'. This is clearly not enough on its own - A more helpful message is required.

Either an external file containing error number descriptions, or a hard-coding within the program of such error messages. Hard coding the errors is not attractive for reasons outlined in section 4.5.7.5. An external file is therefore necessary which contains an explanation of the error. When an error occurs, the file should be referenced, the error text located, and displayed. The system may then shutdown or continue as appropriate, but the user has been informed as to a specific problem.

One particular error needs particular attention - What should occur if the error file is not located. Such an error is clearly problematic, as an error cannot result in a call to the error file which caused the error in the first place! The solution is to hard code the error handling and reporting for this particular error within the program. This error is clearly of the fatal kind.

## 4.6 Conclusion

This section has discussed significant issues relating to the implementation of the system design. The section has taken a low-level approach, with little recourse to actual coding issues. Such issues are discussed in more detail in Appendix A, which contains a routine by routine description of the operation of the system.

---

<sup>26</sup>This would result from leaving the Borland Pascal environment to handle errors. Such errors clearly are of no interest to the user.

## 5. Testing

*Oh! Dreadful is the check - intense the agony -  
When the ear begins to hear, and the eye begins to see;*

*The Prisoner*  
Emily Brontë (1818 - 1848)

### 5.1 Introduction

The implementation of any system must be followed by an analysis as to the extent it satisfied the design requirements. This section will provide this analysis by means of testing the system.

Two forms of testing are necessary: Component and System testing. Component testing occurs before the units are combined to form the entire system. System testing tests the system as a whole, and the degree to which it satisfies the original requirements.

In a system of this size and nature, component testing is exercised during the development of the system. Only when the component performs the operation correctly should it be used. Problems, or 'bugs', that occur within the depths of the components, and their sub-components are likely to become significant when used extensively within the system. System testing as a whole therefore can also be shown to test composite components.

Testing the components of a system cannot easily be demonstrated. A test harness is usually constructed which exercises a particular feature, or module as a whole. The results from the test harness affirm or disprove the belief that the module is working correctly. The development of LEAP takes this approach, yet reached a point when it became impractical. The system had been brought together to a great extent. The testing at this stage used the current interface. For example, the resulting relations from an operator test were compared to anticipated results.

Testing the high level functionality of a system such as this can draw upon example queries that are presented in reference material available. Once the query has been expressed in the defined sublanguage, and providing the necessary relational structures are available, the result of the query can be compared to the published 'result'.

A completed test would produce identical relations, an incomplete test would produce a relation that is not comparable.

The extent to which testing should continue is one of computing's great unanswered questions. There is no specific, correct, amount of testing. It depends on the system and the requirements. It is very much a matter of collective judgement on the part of the development team. The testing will not enter into excessive repetition of simple operations.

The testing within takes place in two stages. The first stage will demonstrate the operators on their own, not combined with other operators. Essentially this is high level unit testing. Following this, the operators will be combined to perform more in the way of testing, but interaction between queries will be demonstrated. This both further demonstrates the operation of operators, and demonstrates the principle of relational closure as implemented.

## 5.2 Test database

The primary test database is taken from STA90. This consists of four relations: book (5.2.1); subject (5.2.2); index (5.2.3); auction (5.2.4). Operations are to be carried out on one or more of these relations and where possible compared to the result published in the text.

A number of operators are not discussed in the published example operations (union, intersect, or product). Therefore a smaller database is taken from DAT90, this consists of two relations A and B. These relations are necessarily union-compatible.

Some operations use other relations from STA90, these have been included in the test result section.

### 5.2.1 Book

Reference	Author	Title
R003	JOYCE	ULYSSES
R004	JOYCE	ULYSSES
R023	GREENE	SHORT STORIES
R025	ORWELL	ANIMAL FARM
R033	LEM	ROBOT TALES
R034	LEM	RETURN FROM THE STARS
R036	GOLDING	LORD OF THE FLIES
R028	KING	STRENGTH TO LOVE
R143	HEMINGWAY	DEATH IN THE AFTERNOON
R149	HEMINGWAY	TO HAVE AND HAVE NOT

### 5.2.2 Subject

Class	Class Name
C1	FICTION
C2	SCIENCE FICTION
C3	NON-FICTION
C4	SCIENTIFIC
C5	POETRY
C6	DRAMA

### 5.2.3 Index

Author	Title	Class #	Shelf #
JOYCE	ULYSSES	C1	12
GREENE	SHORT STORIES	C1	14
ORWELL	ANIMAL FARM	C1	12
LEM	ROBOTS TALES	C2	23
LEM	RETURN FROM THE STARS	C2	23
GOLDING	LORD OF THE FLIES	C1	12
KING	STRENGTH TO LOVE	C3	24
HEMINGWAY	DEATH IN THE AFTERNOON	C3	22
HEMINGWAY	TO HAVE AND HAVE NOT	C1	12

### 5.2.4 Auction

Reference	Date-Bought	Purchase Price	Date-Sold	Sell-Price
R005	17-03-84	25	23-09-86	12.25
R020	02-12-43	4	17-10-88	145.50
R022	09-11-79	7.50	21-11-88	3.25
R048	15-05-68	3.50	16-03-89	8.50
R049	15-05-68	3.50	16-03-89	8.50
R073	21-02-76	18.50	25-03-89	9.25

### 5.2.5 A

S#	Sname	Status	City
S1	SMITH	20	LONDON
S4	CLARK	20	LONDON

### 5.2.6 B

S#	Sname	Status	City
S1	SMITH	20	LONDON
S2	JONES	10	PARIS

## 5.3 Individual operators

The individual operators must be demonstrated. The sample databases available in STA90 are extensive, and examples with expected results published. The print operator demonstrates the data contained within the relations.

Test results are included in Appendix B.

### 5.3.1 Print

The print operation is not an algebraic operation, but crucial nonetheless for displaying the data contained within a relation.

### 5.3.2 Project

The example given projects the authors from the book relation. It is the answer to the query “Who are the authors of the books kept in the library?”.

The operation:  $R1 = \text{project}((\text{book})(\text{author}))$  is executed.

All duplicated author names (joyce, lem, and hemingway) are removed, such that only one tuple exists for each author.

### 5.3.3 Union

The operation:  $UN = \text{union}((A)(B))$  is executed. All duplicate items are removed. (This should be Smith, Clark and Jones’ data). (See DAT90).

### 5.3.4 Intersection

The operation:  $dati = \text{intersect}((A)(B))$  is executed. All items that appear in both relations are placed in the resulting relation (This should be simply Smith’s data).

See DAT90.

### 5.3.5 Difference

The operations:  $datd1 = \text{difference}((a)(b))$  and  $datd2 = \text{difference}((b)(a))$  are executed to illustrate that the operator is position sensitive.

The relation  $datd1$  should contain suppliers located in London, and who do *not* supply part P1 (Clark, London).  $datd2$  should contain all the suppliers who supply part p1, and are not located in London (Jones, Paris).

See DAT90.

### 5.3.6 Product

The Cartesian product of two relations is executed with the operation:  $prd = \text{product}((a)(b))$ . All possible combinations (without duplicates) of the tuples from relations A and B should result.

### 5.3.7 Divide

Earlier, it was mentioned that it is possible to implement the `divide` operator utilising the other operations, namely `project`, `difference` and `product`. A program is constructed which performs this operation:

```
a1=product((lc)(q))
```

```
a2=project((a1)(lecturer,subject))
```

```
a3=difference((a2)(lc))
a4=project((a3)(lecturer))
```

The result, when compared to the published result, is correct. However, LEAP has been implemented such that attribute name differences result in a failure of a union compatibility check. This should not be the case. In order to demonstrate the example in STA90, the underlying relation for the result of a2 has to be altered such that the attributes are lecturer and course, *not* lecturer and subject. This illustrates the benefit of testing, but is not really a significant problem. It is a simple matter to resolve, through provision of some form of renaming command, for relation names, and attribute names.

### 5.3.8 Select/Restrict

The algebraic operator `restrict` has been incorporated into the `select` operator. The example given in STA90 for `restrict` is therefore appropriate:  
`r3=select((auction)(sell_price>purchase_price)).`

The result will differ from the published answer. The reason for this is that the published answer uses integer comparison, whereas LEAP compares the string values. This is because only the string data type has been implemented. The answer is still correct given the data in the database, but not what a user would anticipate.

Another example would be the execution of:

```
r4=select((index)((author='hemingway') and (class='c3'))).
```

This produces the same answer as published.

### 5.3.9 Join

The join operator is tested with the execution of:

```
rj=join((subject)(index)(subject.class=author.class)),
which should list the class name of each authors book.
```

When combined with `select` and `project`, the operator is more useful.

## 5.4 Combined operations

### 5.4.1 Introduction

A series of queries present in STA90 are presented, using the published answer.

### 5.4.2 Project/Divide/Project

Example 4.1 from STA90 proposes the query “Which subjects are not represented in the library?”. The query can be expressed as:

```
r2=difference((project((subject)(class)))(project((index)(class))))
```

The result is the same as the published answer.

### 5.4.3 Select/Join/Project

Example 4.5 from STA90 proposes the query “Find all non-fiction books giving their authors and titles”. The query can be expressed as:

```
s1=select((subject)(class_name='non-fiction'))
s2=join((s1)(index)(s1.class=index.class))
s3=project((s2)(author,title,shelf))
```

The result is the same as the published answer.

### 5.4.4 Select/Project

Query 4.3.3 from STA90 should find the Sell-Price and Cost-Price of all goods delivered to branch L1, and still in stock:

```
p3=select((stock)((branch='L1')and(date_out='INSTOCK'))
p4=project((p3)(sell_price,cost_price))
```

The result can be checked against the tables, and shown to be correct.

### 5.4.5 Restrict/Project/Join/Project/Join

Query 4.3.4 from STA90 should find the producer, product-code and description of all goods sold on the same day they arrived at any branch:

```
R1=SELECT((STOCK)(DATE_IN=DATE_OUT))
R2=PROJECT((R1)(BRANCH,STOCK))
R3=JOIN((R2)(DELIVERY)((R2.BRANCH=DELIVERY.BRANCH) AND
(R2.STOCK=DELIVERY.STOCK))
R4=PROJECT((R3)(PRODUCER,PRODUCT_CODE))
R5=JOIN((R4)(GOODS)((R4.PRODUCER=GOODS.PRODUCER) AND
(R4.PRODUCT_CODE=GOODS.PRODUCT_CODE))
```

The result, when checked by hand, is correct.

### 5.4.6 Select/Join/Project/Join/Select/Project

Query 4.3.5 from STA90 should find the branch#, size, colour and sell-price for all dresses that have not yet been sold:

```

Q1=SELECT( ( GOODS ) ( DESCRIPTION= ' DRESS ' ) )
Q2=JOIN( ( Q1 ) ( DELIVERY ) ( ( Q1 . PRODUCER=DELIVERY . PRODUCER )
      AND ( Q1 . PRODUCT_CODE=DELIVERY . PRODUCT_CODE ) ) )
Q3=PROJECT( ( Q2 ) ( BRANCH , STOCK ) )
Q4=JOIN( ( Q3 ) ( STOCK ) ( ( Q3 . BRANCH=STOCK . BRANCH ) AND
      ( Q3 . STOCK=STOCK . STOCK ) ) )
Q5=SELECT( ( Q4 ) ( DATE_OUT= ' INSTOCK ' ) )
Q6=PROJECT( ( Q5 ) ( BRANCH , SIZE , COLOUR , SELL_PRICE ) )

```

The result, when checked by hand, is correct.

### 5.4.7 Nested Project/Select

The answers to question 4.1.a in STA90 is given as:

```
PROJECT( ( SELECT( ( EX_BOOK ) ( PNAME= ' PITMAN ' ) ) ) ( TITLE ) )
```

Which should result in all titles published by Pitman. The result, when checked, is correct.

### 5.4.8 Nested Project/Join/Project/Select

The answers to question 4.1.b in STA90 is given as:

```
PROJECT( ( JOIN( ( PROJECT( ( SELECT( ( EX_BOOK ) ( PNAME= ' MIT PRESS ' ) ) ) ( ANAME ) )
      ) ( EX_AUTH ) ( ANAME=EX_AUTH . ANAME ) ) ) ( SPECIALISM ) )
```

Which should result in the specialisms of all authors publishing a book with MIT Press. The result, when checked, is correct.

## 5.5 Additional operations

### 5.5.1 Indexing

Indexing has to some extent been implemented. An index may be specified, built and retrieved.

#### Specification

The specification of an index takes to form: `specidx name attribute1 attribute2`, for example:

```
specidx book title
```

An index is built on the relation `book`, using the specified attributes

### Retrieval

An index once built, can be displayed through the command `idxprint name`. For example:

```
idxprint book title.
```

The indices contents are printed appropriately.

## **5.6 Stress**

### **5.6.1 Maximum degree**

The maximum degree of a relation is hard coded as a constant: `MAX_NO_ATTRIBUTES`, and set to 55. However, a limit is likely to be reached in the maximum string that may be entered at the command line, of 128 characters.

When two large relations are combined through a `product` or `join`, an error is reported.

### **5.6.2 Maximum cardinality**

This is difficult to prove, but the implication is that if the system works for a low cardinality, it will work to a high cardinality. However, the implementation of the tuples will encounter errors if the seek position is greater than the maximum value that may be stored in a variable, which is a `word`. The maximum file size that will be properly dealt with is therefore 65535 bytes.

Most read/writes however, read relation tuples sequentially. There is no need to store the file position. The operating system is therefore likely to impose limits before the program itself on the size of a relation.

### **5.6.3 Load**

The system is a single user system. Each operation is executed sequentially. There is therefore no need or ability to test how the system copes under varying loads.

### **5.6.4 Error handling**

Errors are handled as extensively as possible. However, there are always points when an unexpected error occurs. The programming environment handles these through generation of a runtime error. These are not desirable, and where possible are prevented, and coped with internally. Unexpected errors, by their very nature, cannot be brought about in a controlled environment without spending a great deal of time. As far as the author knows, the majority of such errors are handled correctly through testing during the development process.

## **5.7 Miscellaneous operations**

### **5.7.1 Scripts**

Files may be used as command files, or scripts. At start-up and shutdown, special scripts should be automatically executed.

### **5.7.2 Help facility**

An on-line help facility should exist, and give the user a brief overview of the commands that are valid, and the means by which they should be entered. Specific help 'pages' should be displayed, i.e.

Help 1                   - Displays page 1

Help                    - Displays all help.

### **5.7.3 User creation of relations**

The user can create their own relations through use of the `create` command. A relation name is specified, and a series of attributes specified. To create a relation example, with attributes `f1`, `f2` and `f3`:

```
create example f1,f2,f3
```

### **5.7.4 User addition of tuples**

In addition to the creation of relations, tuples may be added through use of the `add` command, and following a prompt.

### **5.7.5 Displaying structures**

The structures within the database may be displayed with `list` (to display the relations), `sources` (to display the script files), and `indexes` (to display the index structures).

### **5.7.6 Redirected output file**

All information that is displayed on the screen through standard output, is mirrored automatically into the specified file, located in `$TEMP$\rardbms.txt`, where `$TEMP$` is the `TEMP` environment variable.

### **5.7.7 Report file**

System occurrences are logged in the report file, which is stored in `DIR_MASK\REPORT_DIR`, as `REPORT_FILENAME`. Debugging may be switched on or off.

### **5.7.8 Report file, with a nested expression.**

The same as 5.7.7, but for a nested expression. The report file shows the progress through the expression. (The report file is taken from the execution of the user defined script in section 5.7.1)

## **5.8 Internet release**

As part of the testing process, and also out of personal curiosity, executable versions of LEAP were placed onto anonymous FTP servers attached to the Internet. A note was placed onto a number of usenet news groups announcing its release, inviting interested individuals to download the system and experiment with it.

It was hoped at the time that a number of people would use it extensively, and bring back a report of their encounters. Logs from one of the ftp servers showed that in excess of 40 people from around the world downloaded it.

Correspondence was received from a few individuals, from the United States of America, New Zealand, Denmark, as well as the United Kingdom. Unfortunately, there was little in the way of correspondence relating directly to LEAP itself. A number of individuals have been asked to be kept informed of its development, and a number have asked for a copy of this documentation when complete.

Whilst it is clear that few problems were reported, it cannot be assumed that people were necessarily happy with LEAP. It is suspected that a large number of individuals downloaded the system thinking it offered a usable system for commercial use, despite extensive warnings to the contrary in all 'publicity'.

One of the correspondents was a University lecturer in a computing department. It is certainly possible that given their interest in the system, the idea of LEAP as a learning tool is not an academic conclusion, and could prove to be a real occurrence.

In summary, the internet release was a useful experience. It is hoped that a few individuals will be sufficiently interested in the system to use it in a wider context, and perhaps resolve a large number of so far undiscovered problems with LEAP.

## **5.9 Conclusion**

Having examined each of the operations separately, and shown that they work satisfactorily through comparison to anticipated results, there is reason to conclude the system works correctly. By combining the operators in a number of different fashions to query published databases, and compare LEAP's results with the published results, the conclusion has to be that the system works as necessary, to the given requirements.

The system has been shown to be able to handle anticipated stress, where possible in an environment of this size.

Therefore, the system works correctly under anticipated stress, and is believed to be able to handle satisfactorily, unanticipated occurrences.

## 6. Conclusions

*“Quidquid agas, prudenter agas, et respice finem.”*  
- Whatever you do, do cautiously, and look to the end.

Anonymous, *Gesta Romanorum*

### 6.1 Introduction

This section will attempt to draw objective conclusions with regard to the success of the project, and the extent to which the design has been implemented. Particular merits of the design/implementation will be highlighted, as will any significant shortcomings. A number of possible enhancements to the system are also discussed.

### 6.2 Merits

LEAP is not claimed as solving all the problems facing database users. It is comparatively short of features compared to commercial, or large scale academic projects<sup>27</sup>. It was never conceived as being on the scale of such systems. It does, however, offer basic functionality for a core component of relational theory. Particular merits of the design are highlighted.

#### 6.2.1 Interface

The interface offered by LEAP is not in the same league as the graphical interface that adorns Microsoft Access. It was a design decision to offer a command line interface, for the relational algebra is procedural in concept, and a command line interface is appropriate in this context.

The interface that LEAP allows all of the relational algebra operators to be expressed unambiguously. Expressions may be nested, and the results of expressions assigned to new relations. Through this process, large ‘programs’ may be executed which result in a relation containing the result of some query. The user is warned of any errors that may be encountered by the system whilst dealing with the expression.

Through a formal BNF definition of the language, the reader should be able to grasp how to formulate queries on the database<sup>28</sup>.

#### 6.2.2 Extension framework

LEAP was, from the outset, not anticipated to offer everything users may require. As a result, the system has been designed to allow other programmers to attach new or

---

<sup>27</sup>Such as Ingres (both university and commercial versions), Postgres, Requiem etc.

<sup>28</sup>Assuming, of course, that the reader understands Backus-Naur-Form for the definition of language!

modified code to the system, which remains transparent to higher levels. New functionality can be inserted into the system, and brought into use.

This feature thereby ensures that LEAP can be extended to cover new areas. Some of the relational operators have a variety of incarnations and all may use different underlying algorithms to achieve the same result. These can be modified and enhanced through such a framework to suit the users anticipated needs.

The extensive modularity and abstraction of LEAP lends itself to a clearer implementation than might otherwise have occurred.

### **6.2.3 Requirements**

LEAP has been implemented on what is probably the most common popular computer environment, namely IBM compatible PC, with MS-DOS. It does not require large amounts of memory, or secondary storage.

### **6.2.4 Completeness**

LEAP offers the complete relational algebra. Examples have been taken from various texts, and shown to produce the correct results. This could make LEAP a useful learning tool. The internet release gave some credence to this conclusion.

## **6.3 Shortcomings**

Completely solving a problem in one attempt is a goal all problem solvers aspire to, and very few achieve. Software development is a very intricate process, and large pieces of software rarely solve every problem completely.

LEAP could easily have been overly ambitious. At the outset, before the design of the system was properly conceived, ideas such as an SQL query processor, and multi-user facilities were considered. These ideas were eventually reduced the much smaller scope that is present in the design, with the possibility of enhancement available through a strong structural framework.

Because of the limited scope, LEAP will not find itself in day-to-day use as the RDBMS of choice. The relational algebra query language will see to that. LEAP has its feet firmly in the theory which underlies RDBMS.

With regards to the features offered:

- LEAP has a poor secondary storage philosophy. In the pursuit of a simpler implementation, and to enable the user to modify data as necessary, text files were used rather than a more efficient and faster binary approach. Once LEAP offers the necessary functionality to remove the possibility of users needing to edit the files, a binary approach should be implemented.

- The BNF definition of the relational algebra requires brackets to be used extensively. This leads to confusion, and a more appropriate language definition that more closely resembles the structures in the various textbooks is definitely desirable.
- The nested loop algorithm underlying the `join` operation is not optimal, especially considering hashing and indices have been implemented. However, the lack of time prevented a wider scale incorporation of the enhancements.
- Hashing tables are not stored between runs. A significant period of time is spent within each operator building hash tables. If they were stored and maintained, the speed of operations would be significantly improved.
- Attributes can be duplicated within a relations header. This can result in the user being unable to specify to the system which specific attribute is required. The solution would be, when a duplication occurs, to incorporate the original relations name to some extent in the new name, e.g. “`rename.attrname`”, as used in systems such as Oracle.
- The system uses capitalisation for the storage of all data. This is restrictive for any significant use of the system, as capitalisation may be an important issue to the user. This would be a simple issue to redress, by removing all conversion that currently occurs at the input line.
- The String data type is the only data type that LEAP supports. This has been shown to cause problems with comparisons, and should be addressed. LEAP has the facility for a large number of data types (number, dates etc.) to be incorporated.

## 6.4 Enhancements

The system has been shown to offer the core functionality required for a relational database management system for querying usage. It does not support the insert-update rules defined by COD79, and therefore cannot be termed truly relational. Yet few commercial systems fully support relational integrity<sup>29</sup>.

A number of enhancements are discussed in this section.

### 6.4.1 Optimisation

Optimisation is a critical component of commercial RDBMS. Users are not expected to consistently enter queries in the most optimal form. The optimal form can depend on so many factors, especially the implementation approach, that it is simply not feasible.

---

<sup>29</sup>Oracle, one of the most successful and popular commercial RDBMS, has only with the latest release of its server software (Oracle 7), supported referential integrity.

However, a number of rules have been devised which are proven to reduce the amount of effort that the system must expend to reduce the resources required to resolve a particular query. These include: Combination of a cascade of projection/selection operations; Commuting of selection and projection; Executing Selection before join/Cartesian product etc. (DES90)

The system as implemented does not have provision for an optimisation module: The query is executed as the system processes each component of the query. In order to be able to optimise the query, an additional step is necessary. This step would produce an internal representation of the query, which the optimisation module may process to produce a more optimal expression if appropriate.

In order for this to be implemented, the command line interface module would pass the command to a new module. The new module would first perform the step known as *lexical analysis*, which involves analysing the source statements and recognising and classifying the various tokens<sup>30</sup>. It entails the discarding of layout characters<sup>31</sup>, which may occur between any two tokens. Furthermore it involves the recognition of *reserved symbols*, such as the keywords and operators used in the particular language being translated. It is the purpose of the lexical analyser to insulate the parser from the lexeme<sup>32</sup> representation of the tokens.

Following lexical analysis, the parsing process follows, which will involve translating the classified structure into a series of calls to functions. Of the two methods discussed earlier in the interface section, the recursive descent approach was implemented. The more complex approach is discussed. This will involve first checking that the query is valid<sup>33</sup>, and that there exists no ambiguity. A representation of the query is built internally. This will take the form of a tree, following the nature of expressions. Each sub-expression within a query will be represented as a further sub-tree in the overall structure (see Figure 3-9). The means by which this process is implemented is discussed extensively in WIR76.

Given that the parsing does not produce any errors (which would be reported to the user, and further processing aborted), the optimisation step can take place. This is not discussed in texts such as WIR76, for optimisation is not appropriate in numerical expression analysis, nor is it appropriate in natural language processing. This is because a different interpretation of an expression can result in a different end result<sup>34</sup>. Within relational expressions, the correct application of optimisation should not result

---

<sup>30</sup>A *token* can be thought of as the fundamental building blocks of the source language being utilised. Consequently, tokens are sequences of characters having a collective meaning. (PAP89)

<sup>31</sup>Such as blanks, tabs, newlines and comments.

<sup>32</sup>The *lexeme* is the sequence of input characters that make up a concrete token. (AHO85, cited in PAP89)

<sup>33</sup>This may take place within the lexical analyser, for example if a token is not recognised as valid.

<sup>34</sup>For example, the lack of brackets:  $7 + 3 * 5$  - Is the answer 50, or 22?

in a different end-result, but should result in a more efficient execution of the expression.

The constructed parse tree is likely to therefore be modified, without loss of meaning. Such modifications are discussed in extensive detail in both DES90, and DAT90.

It is likely, due to the large number of optimisation steps that can take place, that a variety of ‘optimal’ forms may exist. The advanced optimiser should analyse each of the possibilities generated, and choose the cheapest<sup>35</sup> approach. All other parse trees should be disposed of. It is of course true that the process of optimisation *itself* will possibly take longer than simply executing the query, therefore some earlier step should be introduced in the advanced optimiser to take stock of this problem. It should determine whether to proceed, or simply execute the query. This would possibly involve use of algorithms to determine roughly how inefficient an expression is.

The execution of the query will involve processing the resulting tree, and resolving sub-expressions. The result of sub-expressions is passed ‘up’ the tree to the higher level, and further sub-expressions executed. The end result of the process will be a relation, which contains the result of the query.

This entire approach is complex, and the subject of books (such as AHO85). There are other means of optimisation which will not necessarily involve the construction of new modules. The relational operations themselves form the core of the system. It is certainly not true within LEAP, that they are in the best possible form. The best possible form is likely to vary according to the degree/cardinality of relations, and the implementation environment. It is certainly possible to implement more efficient operators, especially the frequently used `select`, `join` and `project` operations. A number of papers discuss the possibilities of different algorithms for such operators, and have been briefly discussed in the system design section<sup>36</sup> (section 4.5.3).

## 6.4.2 Expressions

The implemented language is a prefixed form of the relational algebra. It utilises brackets extensively. An extension that should be seriously considered would be the rewriting of the system to develop an *infix* language which does not rely heavily upon bracketing.

The result would be a more conducive to students of relational algebra. The process of converting expressions published in texts such as DAT90, and DES90 would be less arduous, and have greater parallels.

An infix language would not necessarily be as simple to implement, as the operator is not known before the operands, but a determined effort to implement a two stage parser should reduce this problem by incorporating the requisite.

---

<sup>35</sup>Cheapest in terms of disk operation, CPU utilisation etc.

<sup>36</sup>DES90 provides a large number of references on the subject of optimisation.

### 6.4.3 Indices and relational integrity

The issue of indexing has been raised earlier within the document, and has been partially implemented. Yet there was insufficient time to incorporate the facilities into the system as a whole. Indices on relations will be of direct use within the process of implementing more efficient operators.

An attempt was made to use indices within the one pass join operator, without success. This was because tuples were not being updated in the index structure on disk.

A maintained index structure will directly facilitate the support of primary and foreign keys on relations. Prior to the insertion of a tuple into a relation, a search on appropriate index structures will determine whether duplicates exist. Prior to the deletion of a tuple, a search on the structures will also determine if relational integrity is to be maintained. This will of course add to the overheads of the system, but ensure that the system can be considered truly relational in COD79's definition.

The core of the implementation has already taken place. It simply remains for more work to be done on incorporating the facilities into the overall system.

The first step would be to ensure that when new tuples are added to the relation, the index structures are immediately checked and updated if appropriate. When LEAP is shutdown, the entire relation structures should be written to secondary storage to ensure that the next execution of the system results in complete indices.

The second, and most significant step, would be the implementation of primary and foreign keys. This would require each relation to have a defined primary key, in which no duplicate entries may be found. Directly following this, user defined keys may be implemented. Through this a number of indices would exist on each relation.

Thirdly, the implementation of support for relational integrity. This would necessitate the definition of foreign keys within and between relations. This would not necessarily require new indices to exist on relations, but would not be truly feasible without<sup>37</sup>.

Given the above implementations, the system could then be termed truly relational.

A final step, which relates largely to the optimisation discussion in section 6.4.1, is the incorporation of indices within the existing relational operations. Operators such as `select` could be rewritten to make use of the index structures. A seek for the start of the tuples matching the search criteria, i.e. attribute  $x > 1000$ , the first tuple  $= 1000$  would be found, bypassing the need to search through tuples  $< 1000$  as currently. Having this would require a more complicated expression analyser as a whole, for an `or` operator would cause problems with respect to locating the first tuple.

---

<sup>37</sup>It would be possible to implement relational integrity maintenance *without* indices at all, but this would necessitate the searching of relations in their entirety. This would be very expensive in terms of processing time, and result in a very inefficient and slow system, possibly to the point that the system is unusable.

## 6.4.4 Portability

Pascal is a well known general purpose language that was the most appropriate language for implementation. However, Borland Pascal digresses from the ISO standard for Pascal. Its portability to other environments is non-existent.

If portability to other environments became desirable, then an implementation is required in a language which is more portable, such as C. Implementation in ANSI standard C would ensure that any environment with an ANSI C compiler could execute the program. However, implementation completely within the bounds of a standard would not facilitate a particularly efficient system. Therefore, the lower level operations should be written to take advantage of language implementation features that do not exist within standards, and machine specific features.

Because of the abstraction that has taken place within the design and implementation of LEAP, this process should not be particularly arduous. The high level operations, dependent solely upon the lower level operations and language standards, should not need modification<sup>38</sup>.

## 6.4.5 Functionality

LEAP's functionality has been shown to cover the entire relational algebra. However, there is a need to provide additional functionality to maintain the database. A number of features are missing from this functionality that should at some stage be added:

### 6.4.5.1 Renaming

It is not currently possible to rename a relation, or an attribute within a relation without resorting to editing the data files. This functionality is likely to be necessary if the system were to be used by others. A relation may be created with a random name, and the user decides that they wish to maintain it in the permanent database. It is possible to change the temporary status of a file, but not to change the name.

### 6.4.5.2 Tuple deletion

It is currently not possible for a user to remove tuples held within the database, at least not within the bounds of the system. A significant addition to the functionality, and one which would not be particularly difficult, would be the addition of a tuple deletion operator.

To some extent, the deletion operator has been implemented with the `select` operator. A new tuple may be created that contains all tuples apart from those that do not match a given criteria. However, this is not particularly efficient. As such, a specific `delete` operator is necessary.

---

<sup>38</sup>Unless some feature of the language changes the functionality of the rewritten function, which is likely to require the dependant higher level operations to be modified to anticipate the changed functionality.

Two approaches are immediately apparent: The first would involve creating a new temporary relation into which all tuples *not* matching the deletion criteria are copied. The source relation would then be deleted, and the temporary relation renamed to the same name as the original source relation. The second approach would modify the data held within the specified relation, and remove tuples directly that match the given criteria. Both of these approaches would require the appropriate indices to be updated accordingly.

Another approach comes to light when the question of indices is brought to bear. The index structure will need to be completely updated for the tuple location index will become invalid if a tuple is removed *before* the tuple in question. Tuples prior to the deleted tuple will not be affected, as their location will remain the same. The complete updating of indices will clearly be expensive in time and resources.

In order to resolve this issue, the possibility of deletion flags can be considered. If each tuple has a flag indicating its deleted state, this can simply be marked TRUE or FALSE as appropriate. 'Deleted' tuples are not used in any way by operators. The problem with this is that relations will only grow in size, even when tuples are deleted. To resolve this problem a database contraction is necessary, whereby all marked tuples are removed from the files, and the indices updated. This would occur at the end of a session in LEAP. In commercial systems, the DBMS would be shutdown and compacted as necessary.

#### 6.4.5.3 Tuple updating

In a similar manner to the deletion of tuples, the updating of values within tuples is a necessity. It cannot be guaranteed that the tuple will have the same value throughout the existence of the database. For example, a database may contain the current retail price of cars. This price will change over time.

The updating of tuples will be similar in ways to the deletion. There are two clear methods at first: The tuple can be copied, deleted, updated and reinserted, or directly modified. In both cases the indices may become invalid. To prevent this, the nodes within all indices must be removed, and new nodes reinserted - The position within the index structure (if a B-Tree is used) is likely to change.

#### 6.4.5.4 Additional functions/operators

DAT90 introduces a number of additional functions/operators that have been put forward by various authorities. These include:

- |           |  |
|-----------|--|
| Extend    | Extend allows the computation of values derived from values within a particular attribute of a relation.   |
| Summarise | Summarise allows the aggregation of attribute-values, using operations such as <i>count</i> , <i>sum</i> , <i>average</i> , <i>maximum</i> , <i>minimum</i> etc. Summarise can be further extended to allow grouping of attributes within a relation, such as average price of all models of car, not necessarily the entire relation. |

Outer Join      Extending the ordinary (or inner) join, tuples in one relation having no counterpart in the other should appear with nulls in the appropriate attribute positions. A number of issues arise when outer joins are considered, which DAT90 discusses extensively.

### 6.4.6 Views

A view is described in DAT90 as a “virtual” relation. A virtual relation is represented within the system not as a relation containing its own distinguishable, separate data, but rather by its *definition* in terms of other named relations. This definition consists essentially of a *named expression of the relational algebra*. A view, seen by the user as a relation, is essentially a query of the database. For example, a view V may be defined as all the publishers of books by George Orwell.

When a view is specified in a query, the system automatically replaces the *name* of the view by the expression that *defines* the view. It works primarily because of the principle of algebraic/relational closure.

In order to implement this, essentially all that is required is some form of pre-processor which determines which ‘relations’ are in fact views, and replace the view name by the appropriate expression. The entire expression can then be executed as any other<sup>39</sup>. This is a very straight forward facility, that will greatly improve the functionality LEAP as a whole.

Problems come to bear when a user attempts to update a view. Clearly the view itself cannot be updated, the data underlying the definition must be. The system must therefore be able to identify the underlying data. This will require the support of primary keys, as it is primary keys that perform the row-level addressing function within the relational model (DAT90).

### 6.4.7 Null values

The issue of null values is an important one. It is likely that information will not be complete within any database for many reasons. It should be possible for the user to specify that information is not available. This is usually achieved through use of a ‘null’ value.

However, the integrity constraints of the relational model requires that no primary key may hold a null value. By the implementation of primary keys, this constraint must be enforced.

---

<sup>39</sup>It is perfectly feasible that a view definition may refer to other views, such that a number of expressions may be contained within the one view name. The process of determining views would have to ensure every view has been resolved.

### 6.4.8 Data types

The implementation of LEAP so far has only involved the support of the String data type. This is clearly insufficient for the modelling of real world data. A number of other data types are clearly required, which should include at least a numeric data type. Other data types that should be considered are dates, times and floating point numbers.

The system has provision for different data types. It simply remains necessary that the system should know how to handle them. This would include modification of select/join condition analyser routines, such that the appropriate comparison is drawn. The examples within testing (section 5) demonstrate how inaccurate string comparisons are when an integer comparison is appropriate.

### 6.4.9 Caching

Again relating somewhat to the optimisation enhancement, the idea of caching is also likely to improve the system's performance. Caching involves the internal storage of structures that are frequently accessed. Primarily this would relate to tuple structures within relations. Each tuple would be retrieved from the caching system, rather than the secondary storage medium, and thereby result in a much faster operation.

In order to facilitate this, the reading and writing of tuples would have to be abstracted by a layer from their current implementation. They would then operate the disk through the caching module. The caching module would determine whether to store the tuple in memory or not. When a tuple is requested from disk, the caching routines would first search the internal cache of tuples, and then if unsuccessful, read from the secondary storage.

This functionality is perhaps not necessary given the current functionality of operating systems. Many, including MS-DOS, offer some form of caching functionality. It would therefore be a functionality that would be repeated at different levels in the machine hierarchy. However, an internal caching feature would be much faster than an operating system feature, for the overhead of converting the external representation of the tuple to the internal representation would still exist.

Combining the two provisions is likely to be the most efficient measure. Some form of internal caching of frequently accessed tuples will speed tuple retrieval somewhat.

### 6.4.10 Data Dictionaries

The concept of data dictionaries is described by DAT90 as "data about the data", or "Metadata". The dictionary, in the case of Oracle for instance, is often integrated into the database it defines, and thus include its own definition. This would appear to be the most sensible approach, such that the user can determine how data is utilised without stepping outside of the bounds of the relational concept.

The provision of such a facility would therefore not necessarily involve additional development. For example, it requires that when a relation is created, the appropriate data dictionary relation is updated to include this.

This would require a formal definition of a number of necessary relations that form the data dictionary. In order to maintain these, a new module could perhaps be created which contains a few routines which are responsible for the maintenance of the data dictionary. All operators which therefore modify the state of the database, and is of concern to the defined data dictionary, should therefore contain calls to these operators ensuring that the data dictionary is correctly maintained.

### 6.4.11 Concurrency, recovery, and security

The concept of concurrency with regard to LEAP is a non-issue at least at this stage. The system runs on a single user system. The same applies to security. If the system were to stretch to a multi-user state, then the entire design would have to be revisited, for a new set of problems would come to light that is very likely to require a fundamental rethink of the philosophy.

Recovery from a serious error, or specified state, within the system is a very important aspect of modern RDBMS. The system should be able to ‘roll-back’ to a previous state, either from an error or a point at which the user decides is ‘safe’ prior to some development in the database. This usually requires some form of transaction log, which either be tracked backwards, or some form of data dump, which can be recovered if necessary. No provision is made for either of these, as they are usually of concern in large multi-user systems. If they were required, it would be a straight forward process (in a system of LEAP’s size), to make a backup copy of the database, which can be ‘recovered’, and also to maintain a log of transactions which can be utilised<sup>40</sup>.

## 6.5 Summary

LEAP has been shown to support a large part of the requisites for a relational database management system. It cannot however be termed *truly relational* according to the terms specified in COD79 because of the lack of support for the relational integrity constraints.

DAT90 further discusses the definition of a relation system, and LEAP can be said to be *relationally complete*<sup>41</sup>, according to the criteria of COD79. Relations are supported, and all operations of the relational algebra are available.

---

<sup>40</sup>The script file functionality is a part way step towards this. A file could be built which tracks every operation, and this utilised as a script file if necessary.

<sup>41</sup>As opposed to *fully*, or *truly relational*.

However, the system does provide a strong framework by which further enhancements incorporating integrity constraints can be built, which would then make a *truly relational* system.

In summary, the design has successfully be implemented, and shown to work, providing the necessary functionality specified in the project scope.

## 7. References and Bibliography

- AHO85 Aho, A.V.; Sethi, R.; Ullman, J.D.; “*Compilers: Principles, Techniques, and Tools*”, Addison-Wesley, Reading, Mass., 1985.
- BOR93 “*Borland Pascal with Objects*”, version 7.0 manuals; Borland International; 1993.
- COD70 Codd, E.F.; “*A Relational Model of Data for Large Shared Data Banks*”, CACM 13, No. 6 (June 1970).
- COD79 Codd, E.F.; “*Extending the Database Relational Model to capture more meaning*”, ACM Transactions on Database Systems, Vol.4, No.4, (December 1979).
- DAT90 Date, C.J.; “*An Introduction to Database Systems - Volume 1*”, 5th Edition, Addison-Wesley, 1990, ISBN 0-201-51381-1.
- DES90 Desai, B.C.; “*An Introduction to Database Systems*”, West Publishing, 1990, ISBN 0-314-66771-7
- GON91 Gonnet, G.H.; Baeza-Yates, R.; “*Handbook of Algorithms and Data Structures - In Pascal and C*”; 2nd Edition; Addison-Wesley; ISBN 0-201-41607-7.
- JAR84 Jarke, M.; Kock, J., “*Query Optimisation in Database Systems*”, ACM Computing Surveys, Vol.16, No.2 (June 1984).
- KOR91 Korth, H.F.; Silberschatz, A; “*Database System Concepts*”, 2nd Edition, McGraw-Hill Inc., 1991, ISBN 0-07-044754-3.
- KRU84 Kruse, R.L.; “*Data Structures and Program Design*”, Prentice-Hall, 1984. ISBN 0-13-196253-1
- MAI83 Maier, D.; “*The theory of relational databases*”, Pitman, 1983. ISBN 273-086-227.
- PAL93 Palmer, S.D; “*Programming in Borland Pascal*”, Sybex, 1993, ISBN 0-7821-1151-3.
- PAP89 Papazoglou, M.; Valder, W.; “*Relational Database Management - A Systems Programming Approach*”, Prentice Hall, 1989, ISBN 0-13-771874-8.
- STA90 Stanczyk, S. “*Theory and Practice of Relational Databases*”. Pitman 1990, ISBN 0-273-03049-3.
- STO76 Stonebraker, M.; Wong, E.; Kreps, P.; Held, G.; “*The design and Implementation of INGRES*”, ACM Transactions on Database Systems, Vol.1, No.3 (September 1976).
- STO80 Stonebraker, M.; “*Retrospection on a Database System*”, ACM Transactions on Database Systems, Vol.5, No.2. (1980)

- STO88 Stonebraker, M.; “*Readings in Database Systems*”, Morgan Kaufmann, ISBN 0-934613-65-6
- STU89 Stubbs, D.F.; Webre, N.W., “*Data Structures with Abstract Data Types and Pascal*”. Brooks/Cole, 1989. ISBN 0-534-09264-0
- WIR76 Wirth, N; “*Algorithms + Data structures = Programs*”, Prentice-Hall, 1976. ISBN 0-13-022418-9
- ULL85 Ullmann, J., “*A Pascal Database Book*”, Oxford University Press, 1985. ISBN 0-19-859642