UNIVERSITY OF SOUTHAMPTON

A Desk-top Information Manager

Jorge B. Bocca

Thesis submitted for the Degree of Doctor of Philosophy

UNIVERSITY OF SOUTHAMPTON

1985

ACKNOWLEDGEMENTS

First and above all, I would like to thank Professor David W. Barron for the helpful and timely comments he made while supervising this work. I am grateful to Mark Wallace who made me aware of many design drawbacks and forced me to rethink innumerable problems that I had believed solved. Special thanks must go to Michael Freeston for his pragmatism when the flow of ideas took me to impossible dreams, Professor Michael Shave and Dr. Maurice Flower for all their support and encouragement. In particular, I would like to thank Carol Kealey for typing this thesis. Finally but not least, I would also like to acknowledge the many comments and discussions with the members of the Computer Studies Department at Southampton University.

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

Faculty of Mathematical Studies

Doctor of Philosophy

A Desk-top Information Manager

Jorge B. Bocca

This thesis describes the principles, the design and the implementation of an information management system.

Because this system is intended for general purpose use, it needs to be flexible to lend itself to multiple uses, portable to take advantage of new hardware, expandable to provide a regulated growth path and above all easy to use. A Desk-top Information Manager (ADIM) was designed and implemented to satisfy these requirements.

The main core of ADIM is a highly efficient interconnectable "desk-top" data base management system. This data base system is of a relational type and provides users of it with an interface based on relational algebra. ALFRED is a family of languages designed and implemented for use with ADIM.

This thesis also discusses efficiency problems in relational systems and their solutions within ADIM, ways of implementing the interconnection of data bases and applications of ADIM.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Goals and Motivations -

In this thesis I advocate the use of the relational model of data for the design and implementation of a personal data base system. Recent research in relational data base systems has produced solutions which rely on large and powerful computer systems. I have concentrated on solutions based on small computer systems. My design concentrates on the use of multiple microprocessors. These processors in conjunction with appropriate algorithms can produce highly efficient personal data base systems. The design gives special consideration to future expansions of these personal units. The aim has been to design interconnectable 'desk-top' data base systems.

Small is beautiful, and certainly this is the case of

all microprocessor based systems. Apart from cosmetic considerations, low cost and friendliness are the over-riding factors in the design of personal computer systems. In this thesis, I attempt to demonstrate the feasibility of designing and implementing a personal data base management system which complies with the above requirements.

I believe that a personal data base management system must be: flexible to lend itself to multiple uses, portable to take advantage of new hardware, expandable to provide a regulated growth path and above all, easy to use.

Data independence and the ability to formulate queries in a non-procedural fashion are the distinctive virtues of the relational model of data [CODD70]. It is because I believe that these virtues provide an adequate base for flexibility, portability, expandability and ease of use, that I advocate the relational model of data for the design and implementation of a personal data base system.

Although many people may argue against a relational approach on the grounds of efficiency, I firmly believe that appropriate optimization techniques once incorporated into the design of a personal data base

system can produce an efficient implementation.

In pursuit of the above goals, I have designed and implemented a personal data base management system based on the relational model. ADIM - A Desk-top Information Manager - is the name of this system. ADIM embodies the design principles and ideas presented in this thesis.

ADIM's requirements of flexibility and expandability demanded an architecture where distributed systems and/or multiprocessor systems could be used advantageously depending on the circumstances. Thus, for the sake of efficiency, a user who starts on a one-site-single processor personal system can progress to a one-site-multiprocessor system. Alternatively, another user might have a need for a growth path leading to a loose distributed system, but owing to financial considerations he/she starts with a one-site personal system. An example of the first case is a data base system for use in an automatic document classification system, where the growth in the quantity of documents to classify is accompanied by a gradual deterioration in the performance of the system. Performance in this case, can be improved by increasing the number of processors in the system. The second case is examplified by a university data base. This data base could have been originally set up in the mathematics department and followed later by

the installation of a data base in the physics department.

A logical expansion can be obtained by interconnecting both data bases into a unified distributed system.

In order to develop ADIM as a highly efficient interconnectable "desk-top" data base system, three problems needed to be solved:

(a) distribution of data traffic between processors/sites;

(b) distribution of data traffic between secondary and main memories;

(c) processors scheduling.

A direct attack on the data traffic bottlenecks, (a) and (b), is to curb the amount of traffic in the most saturated data pathways of the data base system. For example, if we assume parallel processing in a multiprocessor site with several disc units, it would be more efficient to have each processor interacting with a different disc unit rather than several of them interacting with one disc. This is because the disc channel capacity would not be saturated and therefore, no processor would be kept waiting for other processors to finish.

In order to obtain a meaningful and efficient distribution of data onto the different units of secondary storage and throughout the interconnected "desk-top" data base systems, data must be partitioned. Decomposition techniques and data base design tools provide the necessary elements for a solution to problems (a) and (b). Techniques for decomposition of a data base are presented in chapter 5, together with some tools for the design of data bases.

Unfortunately, as a result of decomposing a data base, simple user's queries can become very complex queries. To solve this problem, ADIM makes use of techniques based on optimization by query transformation [PALERMO, PECHERER]. Lastly, but not least in importance, once a relation has been identified for retrieval, this operation should be executed in minimal time. For this, data structures and access methods appropriate to the general nature of data bases should be used. B-trees [COMMER79] are used by ADIM as the unique data structure. This, I feel is a very efficient solution for static and volatile data.

Finally, searching strategies based on the particular data base and the system's architecture produce the solution to problem (c). Cost functions were defined, so that decisions regarding alternative strategies could be

made. A discussion of cost based strategies in ADIM is held in chapter 6.

1.2 Review -

The development of the earliest and possibly most comprehensive relational data base management system to date - INGRES, was completed by a team working at the University of California, Berkeley, under the direction of M. Stonebraker [HSW75]. The proposal of a relational model of data is the contribution of E.F. Codd [CODD7Ø].

Considering the potential areas of application it is not a surprise that research and development of distributed data base systems is a very active field [HELLER, STONENEUH, HEVNER, DEPPE, ADIBA, STOCKER]. Some experimental systems have been implemented [CHAM, HELLER], while others offering many interesting features are being developed. The Polypherne project [ADIBA] in France and the PROTEUS project [STOCKER] in the UK are distinctive examples of this trend. These projects, like similar projects in the USA, are all orientated towards widely distributed computer networks. As far as the author knows, not much emphasis has been placed on the design and implementation of data base systems for small microcomputers sharing a common pool of data. Also, very

very few relational data base management systems capable of running on small microcomputer (personal computers) have been implemented. The existing systems in this class do not exhibit a great deal of sophistication. Perhaps the most popular among these systems is DBaseII [ASHTON]. Other systems in this class are: Data Ease and Condor [JACOBSON].

The distribution of processing power and storage cells has a significant effect on the design of data bases. At the global level of design, a common data model is required. A global model which supports heterogeneous data models at the local nodes is the main feature of PROTEUS [STOCKER].

Another problem of particular importance to distributed data base management systems is the partitioning of data bases into physically distributed files. Interesting results in this area have been reported in recent years [SKCWHC, WUN]. Siang Wung's solution is based upon the use of specialized hardware [WUN].

Because of the efficiency considerations, research into access methods, data storage structures and file organization techniques have received a deserved amount of attention [HS75, HELD75, HELSTO75, LITWIN, TAMMIEN,

YAO, FREDKIN, FAGIN, QUITZOW, LARSON, BAYER, GUDES, BURHARD]. B-trees [BAYER, COMMER79] and Extendible Hashing [FAGIN] have proved to be very efficient schemes for handling very large files in relational data bases.

Recently, interest in data bases for expert systems has grown rapidly. Here, efforts to bring together methods from the fields of artificial intelligence and data base have created an area of mutual interest in both research communities. A number of researchers in recent years have seeked to exploit the similarities between logic based deduction and relational data base concepts [GALLAIRE, KOWALSKI, NICOLAS]. Interesting results have been produced by the use of AI techniques in conceptual modelling [JARVAS] and in the solution of efficiency problems of relational data base systems [JARKE].

1.3 Organization of the Thesis -

I have divided the exposition into eight chapters and eight appendices. Chapter 1 is this introduction, chapter 2 discusses the design principles, chapter 3 presents the users' interface to the system, ALFRED a family of languages, chapter 4 discusses the major architectural features of ADIM, while chapters 5 and 6 cover the details of solutions to the efficiency

problems, chapter 7 is an overview of the implementation of ADIM, and finally, chapter 8 provides concluding remarks and some open problems. The appendices provide demonstrations of ALFRED, utilities for the data base administrator, and complete listings of the more interesting programs in the implementation of ADIM.

CHAPTER 2

ADIM - DESIGN GOALS

ADIM is a relational data base management system, primarily intended for use on microcomputers. This chapter presents a general description of the design goals in ADIM.

2.1 Simplicity -

At the architectural level, simplicity is the predominant theme. I believe that a simple language does not necessarily reduce the power of expression available to users. It might produce longer sequencies of queries, but the queries themselves would not be more difficult to express than in a more complex language. In terms of efficiency, there are advantages in using a simple language: there is no need for sophisticated and bulky software to deal with major query decomposition [WOYU];

the size of the parser for the language is considerably reduced, and so grammatical and semantic analysis is accomplished with savings in space and time; but above all, the application of optimization techniques and the estimation of costs becomes simpler.

The query language in ADIM is of an algebraic type [CODD72], and includes: join, an extended restriction operation, projection, union, relative complement, and aggregate operations such as average and count.

Three different versions of the query language co-exist in ADIM. ALFRED is a family of languages for use with all types of relational systems. ALFRED-U is a language for casual users, while ALFRED-VC and ALFRED-K provide the interface between ADIM and general purpose languages such as 'C' and PROLOG [CLOMEL, RJLK78]. Details of these linguistic variations of the ALFRED language are reported in chapter 3.

2.2 Cost effectiveness and portability -

The imposition of any computer model or operating system would defeat one of the major objectives of ADIM, that of COST EFFECTIVENESS. Portability of ADIM is only restricted by the specification of a minimum hardware

configuration, that is an 8-bit microcomputer system able to run the CP/M Operating System [CPM].

2.3 Modularity -

Modular expandability is implemented in ADIM, not only by allowing the hardware to expand locally, but also laterally, by interconnection of two or more ADIM systems. In this way, the sharing of data by a community of users is possible, i.e. an architectural design for interconnectable 'desk-top' data base units.

2.4 Compactness -

A unique file structure used throughout the entire data base system has not only made implementation simpler, but has also contributed to the production of a more compact and efficient data base system.

2.5 Dynamic data and static structures -

The choice of B-trees as the unique file structure has provided ADIM with the capability of dealing efficiently with volatile data without jeopardizing

performance on more stable data environments. Other types of static structure, significantly ISAM [IBM66] type files, have a tendency to rapid deterioration of performance on volatile data.

But above all, B-trees are one of the cornerstones of an effective costing system. The heavy extra load imposed on the data base system by the collection of statistics for optimization purposes is avoided, and a neat, clean alternative is offered by the use of B-trees.

It is this file structure, used in conjunction with cost functions which provides a basis for run-time optimization in ADIM.

2.6 Efficiency -

The choice of the relational model of data, as a central feature of ADIM, on its own induces significant problems of efficiency, in addition to those already found by the implementation of a non-relational data base management system for microcomputers.

Efficiency is sought in ADIM at three levels: design decisions at an architectural level, during the setting up of data bases by the Data Base Administrator, and

dynamically at run-time.

2.7 Decomposition -

The use of D-join and D-union in the process of decomposition [BOCCA] will normally produce relations with a small number of attributes and relatively small cardinality. ADIM, in consequence, assumes small relation sizes to determine basic optimization strategies. Thus, while many relations may be involved in a query, their relative sizes are small. This approach seeks a maximization of parallelism in the evaluation of queries. Decomposition is discussed in chapter 5.

2.8 Distribution of data and processing power -

An architecture for easy distribution of data and concurrent processing has been sought since the earliest stages of design. This can be seen throughout the system, from the file structures supported at the lowest level of the system up to the data base design tools provided by ADIM.

2.9 Data base design -

The provision of design tools to the data base administrator, not only induces good design of the data base by encouraging normalization, data integrity and security, but also produces small relations to be stored and manipulated by the data base system. This approach does not restrict the diversity of views that can be supported in the data base, on the contrary it encourages a versatile use of views at the highest level. The basic tools provided by ADIM for data base design are: decomposition functions [BOCCA] and enforced use of unique keys.

2.10 Operational overview -

Hypothetically, a user of the ADIM system may enter a query at any node in a network of microcomputers (and indeed, computers in general). Since distribution details are invisible to the user, the query is submitted as if the data base were centralized at the user's node. Likewise, the result of the query is placed at the user's node if not specified otherwise.

Parsing of the user's query is normally done at the entry node. The subset of the data base required to

satisfy the query is determined by a master node. Consultation of the system's catalogue (itself, a set of relations) provides the locations in the network of the required data. Then the query is decomposed into subqueries, which in turn are submitted to remote nodes in the network for processing. At this stage and for a majority of cases, more than one decomposition of the query is possible and several alternative strategies of processing thus emerge. A cost analysis of the different strategies is undertaken, cost comparisons are made and a strategy is selected. This results in sub-queries which are processed by remote nodes. The intermediate relations produced by the remote nodes are composed by the master node into one relation. This relation is finally passed to the entry node which originated the query.

CHAPTER 3

LANGUAGES

3.1 Introduction - ALFRED -

This chapter discusses a group of languages available in ADIM. All of these languages belong to a family, and they provide users of ADIM with facilities to create, maintain and destroy data bases. In addition, once a user starts interacting with an ADIM data base, these languages provide facilities to create, maintain and destroy relations, as well as to query, input, delete and update data on the relations. The query section of these languages are based on a relational algebra [CODD72]. Except for the syntax, the operators of the algebra are the same in all the languages. The group of languages is given the generic name: ALFRED - A Language For RElational Decomposition. The different variations are:

ALFRED-U : ALFRED for Users,

ALFRED-VC : ALFRED with Views and Characteristics,

ALFRED-K : ALFRED for Kernel.

Users of ADIM can interact with their data bases by using any of the three syntaxic variations of ALFRED: U, VC or K. It is expected that casual users would favour ALFRED-U. ALFRED-VC was designed for use by data base administrators, Prolog programmers and in general, the serious users of ADIM. Users interested in using ADIM from their own favourite programming language, can do so by using ALFRED-K.

The subsequent discussion is divided into sections. The first section introduces ALFRED in general, without paying much attention to the syntaxic details of it. Subsequent sections discuss the syntax and semantic of ALFRED. The chapter is closed with a discussion on the rationale for having three different syntaxic versions for ALFRED. As a preamble to the discussion, I should mention here, that although all of the facilities in ALFRED are described in this chapter, I have focused on the query facilities. This is so, because it is in the query sublanguage where the differences among the three versions of ALFRED, are more pronounced.

3.2 ALFRED -

ALFRED is essentially a command language for the manipulation of relational data bases. It does include facilities to create, delete and modify data bases as well as the relations within the data bases. Its retrieval command uses a query sublanguage based on relational algebra, [CODD72], i.e. a collection of operators which deals with whole relations, yielding new relations as a result. The command to delete data from a given relation, uses a subset of the retrieval sublanguage to specify the deletion criteria. The same thing applies to the update command. A discussion on some of the main commands in ALFRED is started below with the retrieval command.

It is obvious that the retrieval power of the query sublanguage would be ultimately determined by the set of algebra operators selected for the sublanguage. Because of this, I aimed to define the query sublanguage, in ALFRED's retrieval command, with a set of operators that is complete in a relational sense [CODD72]. Also, for self-evident reasons, I tried to make the syntax of this sublanguage, easy and efficient to use. That is, the sublanguage should encourage users to produce clear sentences, while at the same time, it should discourage them from using long and convoluted sentences.

R. Pecherer as part of his work in query optimization [PECHERER], proposed four equivalent and complete sets of operators. These sets were:

```
S1 = {
       restriction, product,
       projection, division
     }

S2 = {
       join, projection,
       difference
     }

S3 = {
       restriction, product,
       projection, difference
     }

S4 = {
       join, projection,
       division
     }
```

He also proved that algebraic expressions containing operators taken from any one of these four sets can be mechanically converted to equivalent expressions using operators of any one of the other three sets. For the purpose of selecting a minimum set of operators for ALFRED, a closer inspection of the four sets is undertaken below.

At first, due to their simplicity, the use of product and restriction looked very attractive. However, join

and restriction are very common occurrences in queries. After some consideration, I adopted join and restriction, since product can easily be generated by using join with a condition evaluating to true in all cases. At a later stage and after some practical experiences with ALFRED, I felt that in order to facilitate the construction of ALFRED's sentences, product was a desirable operator. Thus, product was added to the retrieval set.

From the point of view of optimizing the processing of queries, the choice of join and restriction also gave me a greater scope. Once the previous decision was taken, the choice of a set of operators was greatly simplified.

The second choice to be made was between the division and set difference operators. The transformation of an expression using division into an expression involving product, projection and difference is by no means simple. Nevertheless, queries involving division do not occur very often, and although queries involving difference are not very common either, the actual implementation of difference is much simpler than the implementation of division. Thus, I opted for set difference and the retrieval set became:

```
{
  restriction, join,
  projection, difference,
  product
}
```

For the convenience of users and in particular the data base administrator, I included some operators beyond Codd's definition of relational completeness. They are the union and intersection operators, and those data operators normally described as aggregate operators/functions. Examples of the latter are: total, average, max, min, etc.

Two classes of aggregate operators are included: scalar and vector aggregates.

A scalar aggregate when computed gives a single scalar value. For example, one may want to know the average age of all the students attending one particular college.

Vector aggregates differ from scalar aggregates in that they return a set of values. The data to be aggregated is logically partitioned by one or more property(ies), e.g. age, sex, social class, etc. For example one may want to know the average age of students for each social class attending one particular college.

ALFRED-U implements the algebra operators in an interactive query sub-language. This form is for the convenience of casual users of ADIM. Alternatively, by using ALFRED-VC or ALFRED-K, all of the algebra operators mentioned in the preceding paragraphs can be used in an embedded form as function calls in general purpose languages, in particular Prolog [CLOMEL].

The next section introduces some of the retrieval facilities in ALFRED, while specific details of syntax and semantic of ALFRED are presented further on in this chapter.

3.3 An introduction to retrieve -

As its name suggests, ALFRED performs decomposition of queries and composition of results into relations. This feature of ALFRED is completely transparent to casual users of the language. For them, ALFRED-U provides a simple and easy to use interface to their data bases. The data bases themselves, would normally be set up by a data base administrator (DBA). To do this, the DBA will normally use ALFRED-VC, a version of ALFRED that knows about views and characteristics. At this level, the DBA is also provided with a number of data base design tools. The tools, views and characteristics are

discussed at length in Chapter 5. ALFRED-K is a virtual machine for the ALFRED-VC interpreter, and as such, it is hard to use by ordinary users.

As an introduction to the syntax of ALFRED and also to obtain an intuitive feeling for the usefulness, power of expression and general difficulty in using its retrieval facilities, some simple queries, all written in ALFRED-U, are presented below.

RESTRICTION:

The restriction operator chooses those tuples of a relation which satisfy a given condition. For example,

```
RETRIEVE contract WHEN [date>'31/12/81']
        INTO new_contracts?
```

could be interpreted as: those contracts signed after the 31/12/81; put them into the relation new_contracts.

PROJECTION:

This operator in its simplest form returns the specified attributes of the given relation, and

eliminates duplicates from the result. The projection in a query is specified by the INTO part of the query. For example, the query

```
RETRIEVE employee WHEN [salary>10000]
        INTO highpaid ^ [name, dept, salary]?
```

selects the name, dept and salary of those employees earning a salary greater than 10000. It put the data so selected into the relation highpaid. More sophisticated uses of the projection operator in ALFRED, allow the specification of more general assignments of values to the attributes in the result relation. For example,

```
RETRIEVE employee WHEN [dept='production']
        INTO bonus ^ [name, pay=salary*0.1]?
```

gives employees in the production department, a bonus payment of 10% of their salary.

Trivially, at run time, the evaluation of the restriction and projection can be collapsed together into one process, thus eliminating the need for the generation of a temporary relation as well as the file accesses associated with it.

JOIN:

Takes two relations as operands. The result relation is formed by the concatenation of a tuple of one relation with a tuple of the other relation whenever their identifying keys match. In fact, a weaker condition applies, but in most queries, the above condition is sufficient. An example of a query involving join, is

```
RETRIEVE enquiries :*: contracts INTO
                                enq_to_contracts?
```

This query produces as result the relation enq_to_contracts which relates a contract to the original enquiry that led to it.

PRODUCT:

Corresponds to the cartesian product of two relations. An example of its use is given later on, in this section.

DIFFERENCE:

This is the set difference of two relations. This

operator indirectly and in conjunction with product, restriction and projection provides an algebraic counterpart to the universal quantification in a first-order predicate calculus. For example, consider the relation ACCOUNTS[acc_no, currency, amount] which holds information on the type of currency used by customers, and the relation RATES [curr_name, rate] which holds information on the exchange rate of currencies, then the set of queries:

```
RETRIEVE ACCOUNTS INTO T1 ^ [acc_no]?
RETRIEVE RATES INTO T2 ^ [curr_name]?
RETRIEVE T1 (*) T2 INTO T3?      /*product*/
RETRIEVE ACCOUNTS INTO T4 ^ [acc_no, currency]?
RETRIEVE T3 :-: T4 INTO T5 ^ [acc_no]?
                                 /*difference*/
RETRIEVE T1 :-: T5 INTO acc_in_all_currencies?
```

produces the relation acc_in_all_currencies, with the names of those clients who hold accounts in all the currencies in which the company deals. Obviously, this query could have been written in a shorter form.

UNION, COUNT and AVERAGE below, are self-explanatory. In this example, the relation contr holds information about contracts:

```
/*union*/
RETRIEVE contr :+: newcontr INTO allcontr?


/*count*/
RETRIEVE COUNT OF contr ^ [product]
         BY supplier_name INTO qcontr?


/*average*/
RETRIEVE AVERAGE OF contr ^ [amount]
         BY supplier_name INTO av_x_supplier?
```

3.4 Algebra Operators -

This section defines the operators of the algebra in ALFRED. The definitions given by Pecherer for the algebra operators, were modified in ALFRED. This was done in order to achieve a terser syntax for the retrieval sublanguage. Practical usage of ALFRED indicates that no significant differences exists in the power of expression of the two languages. The definition of the operators below, assume some familiarity with the basic concepts of the relational model of data. Further details about this model can be found in C.J. Date's book, [DATE]. The notation used in the definitions is explained immediately after its first use, and in fact, it is based on ALFRED-VC.

3.4.1 Join -

Let R and P be relations. Let r and p be tuples in R and P, respectively. The join of R and P is defined by:

$$R :*: P = \{ rp/K \text{ is a subset of } L \text{ and } r[L] = p[K] \}$$

where

- rp denotes the concatenation of tuples r and p, without duplicate attributes;
- r[K] refers to the set of attributes in the primary key for relation R; and
- r[L] denotes the tuple containing only those attributes specified by the list L.

3.4.2 Product -

Let R and P be relations. Let r and p be tuples in R and P, respectively. The product of relations R and P is defined by:

$$R(*)P = \{rp/r \text{ in } R \text{ and } p \text{ in } P\}$$

3.4.3 Restriction -

Let R be a relation. The restriction of R on predicate [sel-pred] is defined by:

```
R@[sel-pred] = {r/r is in R
                 and [sel-pred] is true
               }
```

where,

sel-pred	is a boolean predicate involving <sel-expr>, the negation ~<sel-expr> and the connectives: AND and OR;
<sel-expr>	is <expr><cmp><expr>;
<expr>	is an expression involving attributes of R, scalar constants and arithmetic operators from the set {+, -, *, /};
<cmp>	is one of {<, <=, >, >=, =}.

3.4.4 Projection -

Let R be a relation and L a list of attributes for R. The projection of R on L is defined by:

$$R\ \hat{}\ [<L>] = \{\ r[<L>]/r \text{ belongs to } R\}$$

where,

<L>	is a list of <l-expr>;
<l-expr>	is <att> = <sel-expr> or just <att>;
<att>	is an attribute in R;
r[<L>]	is the tuple containing those attributes specified by <att> after <l-expr> has been evaluated and the result assigned to <att>. If <l-expr> is just <att>, it is interpreted as <att> = <att>.

3.4.5 Union -

Let R and P be relations. The union of R and P is defined only if R and P are union compatible [see 3.4.5.a], by:

$$R :+: P = \{r/r \text{ is in } R \text{ or } r \text{ is in } P\}$$

3.4.5.a Union Compatible -

Relations R and P are said to be union compatible, if the attributes for R and P are in a one-to-one correspondence such that the corresponding attribute are

defined on the same domain.

3.4.6 Difference -

The difference of relations R and P is defined only if R and P are union compatible [see 3.4.5.a], by:

$$R :-: P = \{r/r \text{ is in } R \text{ and } R \text{ is not in } P\}$$

3.4.7 Intersection -

The intersection of R and P is defined only if R and P are union compatible [see 3.3.4.a], by:

$$R :.: P = \{r/r \text{ is in } R \text{ and } r \text{ is in } P\}$$

3.4.8 Scalar -

Let R be a relation. A scalar x is the single value defined by:

$$X:R[A] = f(r[A]), \text{ for all } r \text{ in } R.$$

where

f(R[A]) is the application of function f to R[A],

x is the user's name for function f,
e.g.: TOTAL, COUNT, AVERAGE, etc.

Remark:

For consistency purposes, ADIM always produces a relation as result (except for errors).

3.4.9 Vector -

Let R be a relation and L a list of attributes for R. A vector F is defined by:

```
F:R[A]/[<L>] = {x = (p[L], f(q[A]))/p is in R ^ [<L>]
                    and q is in R@[L = p[L]] for each p
               }
```

3.5 Other commands -

The commands to create and maintain data bases provided by ALFRED, also have an interactive counterpart. Some of these commands are executed from within an ALFRED session, while others stand as self-contained programs

executable as commands in the host operating system.

A summary description of these commands follow.

3.5.1 mkdev -

This command is used to incorporate a new device or file to ADIM. In order to ensure portability as well as improved efficiency, ADIM does not rely upon the file structure of the host operating system. To accomplish this, a catalogue of devices and data bases available to ADIM is kept in the host file "alldbs". The existence of this mechanism demands of the host operating system a capability to create and maintain sequential files. I do not think that this demand is a restriction in any operating system commercially available. The sequential file is only used for bootstrapping the ADIM system, which in turn, only recognizes its own file structure. Thus, the task of mkdev is to prepare the new device for use by ADIM and to register in "alldbs" that this device is ready for use. For instance, by typing

```
mkdev data 40000
```

the host file "data" will be registered as an ADIM device having 40000 pages of storage capacity.

3.5.2 dbmk -

The dbmk command creates a new data base by building templates for the systems relations and registering the name of the data base in the host file "alldbs". A catalogue of relations in a data base is kept by a set of relations known as system relations.

From the point of view of the implementation of ADIM, the use of relations to describe other relations as well as themselves, has considerably reduced the size of the software to be written. This reduction is possible because of the shared use of software modules between the system and the users, i.e. there is no need to write special software to handle system catalogues [RDBMS, MRDS]. As an example, the command

```
dbmk dept 1
```

will create the data base 'dept' in device 1. This means that the system relations for data base 'dept' will reside in device 1. Users relations for this data base (or any other data base) can reside anywhere in the ADIM system.

3.5.3 dbrm -

It is the counterpart to dbmk. Thus,

```
dbrm dept
```

will release back to the ADIM system all the storage space occupied by the relations in the data base 'dept'. Also, the entry for 'dept' in "alldbs" will disappear.

3.5.4 display -

The issue of the display command will print the named relation in the user's terminal. Display uses a standard form of presentation. Typing

```
display staff
```

will print the relation 'staff' in the user's terminal.

3.5.5 create -

An interactive facility to create new relations. Create provides the user with help in the definition of the primary key [CODD72] for the new relation as well as

asking the user for the name and format of the attributes of this relation. The issue of the command

create staff 1

will initiate a dialogue with the user. This dialogue, ultimately will define the attributes and keys for the relation 'staff'. Once the dialogue is finished (successfully) the appropriate entries will be made in the system relations. Also, storage for 'staff' will be allocated in device 1.

3.5.6 destroy -

It is the counterpart to create. Thus, the command

destroy staff

will eliminate the relation 'staff' from the system relations and will also release the space occupied by this relation.

3.5.7 append -

Interactively adds a new tuple to a named relation. For instance, the issue of the command

append staff

will prompt the user with the name of each attribute, and then it will use the data so collected to add a new tuple to the relation staff.

3.5.8 delete -

Deletes those tuples in a named relation. The tuples deleted are those which satisfy a given condition. The condition is specified by using a subset of the language used for retrievals. The syntax of the command follows our own previous notational definitions, and specifically is denoted by:

delete <relation> WHEN [<sel-exp>]

For example,

DELETE staff WHEN [age>65]?

remove from the relation staff all the members of staff that are older than 65 years of age.

3.5.9 update -

Updates data in a given relation. Its syntax is similar to the delete command. More formally, it is denoted by:

```
update <relation> WHEN [<sel-expr>] INTO [<L>]
```

An example of update is

```
UPDATE staff WHEN [dept = 'production']
             INTO [salary = salary * 1.1]?
```

which gives members of staff in the production department an increase in their salaries of 10%.

In general, the syntax of non-retrieval commands does not differ very much among the tree versions of ALFRED: U, VC and K. Hence, the discussion on the specific syntax of these commands is postponed to chapter 5, where the relevant syntaxic details are discussed as part of a more general discussion on the implementation of ALFRED. Thus, for the remainder of this chapter, I concentrate on

the syntax of ALFRED's retrieval facilities.

3.6 Syntax of ALFRED-U -

The syntax of ALFRED-U is presented in this section. The syntax for ALFRED-VC is a derivation of ALFRED-U, and in fact, most of it has already been presented. It was used as the notational device to explain the semantic of the retrieval algebra, in section 3.4. The explanation on syntax of ALFRED-K is postponed to Chapter 5, where it is explained along with the details for the implementation of the ALFRED interpreter. The same applies to the non-retrieval commands of ALFRED-VC.

The notation used to define the syntax of ALFRED-U is based on a derivation of BNF notation. Non-terminal tokens are enclosed by < and >. Curly brackets are used to represent an optional repetition (Ø to n times). The description of the syntax follows:

ALFRED-U SYNTAX

```
<Ucomms>      ::= {<Ucomm>} OFF  /*logout ALFRED-U*/

<Ucomm>       ::= <retrieve> '?'
                | <delete>   '?'
                | <update>   '?'
                | <copy-str> '?'
                | <rmrel>    '?'
                | <mkrel>    '?'
                | <rmdb>     '?'
                | <mkdb>     '?'
                | <mkdev>    '?'
                | <display>  '?'
                | <append>   '?'
                | <logindb>  '?'

<retrieve>    ::= 'RETRIEVE' <relexp> 'WHERE'
                                {<restrcond>} 'INTO'
                                RELATION <projlist>
                | 'RETRIEVE' <aggregate> 'OF' RELATION
                             <projlist> 'INTO' RELATION
                | 'RETRIEVE' <aggregate> 'OF' RELATION
                             <projlist> 'BY' ATTRIBUTE
                             'INTO' RELATION

<relexp>      ::= RELATION
                | RELATION <dop> RELATION

<dop>         ::= :+:      /*union*/
                | :*:      /*join*/
                | (*)      /*product*/
                | :.:      /*intersection*/
                | :-:      /*difference*/

<restrcond>   ::= '['<selexps>']'

<selexps>     ::= <selexp>
                | <selexps> 'AND' <selexps>
                | <selexps> 'OR' <selexps>
                | '~' <selexps>      /*not*/
                | '('<selexps>')'

<selexp>      ::= rexp 'AND' rexp
                | rexp 'OR' rexp
                | '~' rexp       /*not*/
                | '('selexp')'
```

```
<rexp>        ::= dexp '<' dexp
                | dexp '<=' dexp
                | dexp '=' dexp
                | dexp '>' dexp
                | dexp '>=' dexp
                | sexp '<' dexp
                | sexp '<=' sexp
                | sexp '=' sexp
                | sexp '>' sexp
                | sexp '>=' sexp

<dexp>        ::= NUMBER
                | <dattrib>
                | dexp '+' dexp
                | dexp '-' dexp
                | dexp '*' dexp
                | dexp '/' dexp
                | '-' dexp
                | '('dexp')'

<sexp>        ::= <sattrib>
                | STRING    /*string of characters*/

<dattrib>     ::= DREG      /*a numeric attribute*/

<sattrib>     ::= SREG      /*an alphanumeric attribute*/

<projlist>    ::=           /*empty list*/
                | '^' '['<projspec>']'

<delete>      ::= 'DELETE' RELATION 'WHEN' <restrcond>

<update>      ::= 'UPDATE' RELATION 'WHEN' <restrcond>
                  'INTO' '['<projspec>']'

<projspec>    ::= <assign>
                | <projspec> ',' <projspec>

<assign>      ::= ATTRIBUTE
                | DREG '=' dexp  /*DREG - numeric
                                   attribute*/
                | SREG '=' sexp  /*SREG - alphanumeric
                                   att*/

<copy-str>    ::= 'COPY' 'STRUCTURE' RELATION 'TO'
                      RELATION  /*replicate structure*/

<rmrel>       ::= 'DESTROY' RELATION

<mkrel>       ::= 'CREATE' RELATION /*interactive
                                      invocation*/

<display>     ::= 'DISPLAY' RELATION
```

```
<rmdb>          ::= 'DBRM' DATABASE

<mkdb>          ::= 'DBMK' DATABASE

<mkdev>         ::= 'MKDEV' DEVICE   POSITIVE-INTEGER

<append>        ::= 'APPEND' RELATION  /*interactive*/

<logindb>       ::= 'LOGIN' DATABASE  /*change current data
                                        base*/
```

3.7 The user and the languages -

In this chapter, a family of languages based on a relational algebra have been presented. ALFRED-U is a language devised for casual users of ADIM and ALFRED-VC and ALFRED-K are languages aimed at more sophisticated users of ADIM. In designing these languages, I have tried to satisfy the different and sometimes conflicting requirements imposed on a language by the two communities of users. In doing so, I had four options open to me:

i) Designing two languages with different roots, but specifically aimed at both types of users. For instance, a non-procedural query language based on a first order predicate calculus for the casual users and a procedural language based on a relational algebra for the more sophisticated users.

ii) Embedding the language for casual users into a general purpose programming language. For instance, an EQUEL [STOROWE] type of solution.

iii) Designing two languages sharing the same roots, but with different external appearances. For instance, two languages based on a relational algebra, the first language with interactive facilities for casual users and the second language with functions called from a general

purpose programming language for the non-casual users. Obviously, both languages supporting the same set of relational operators.

iv) Designing a new general purpose programming language or extending an existing one, so that data base facilities are built into the language.

Although case (iv) has been advocated by research workers as the most positive solution [STOROWE], the scale and scope of this project make this alternative prohibitive. Consequently, I have discarded this alternative.

I believe that case (ii) produces a mismatch between the languages. Confusion to users is caused by the combination of procedural and non-procedural languages. An example of this occurs in EQUEL when a distinction has to be made between use of interactive INGRES and 'G' programs with embedded QUEL statements.

Alternative (i) adds to the problems of (ii), the learning of a new language.

Finally, I compromised and chose an alternative that is basically (iii) with some elements of (ii), as a result of the analysis above. I designed and implemented

ALFRED a family of three languages based on a relational algebra. The algebra fits neatly with the constructs of high level general purpose languages of the type of 'C' and/or PROLOG. ALFRED-U is the interactive language for the casual users and ALFRED-VC and ALFRED-K cover the needs of non-casual users.

CHAPTER 4

ARCHITECTURE

4.1 ADIM - Basic Modules -

ALFRED provides group of users with a mechanism to support a variety of logical views over a common pool of data. Admittedly, this is not a capability unique to ALFRED. A number of other systems provide it as well. I believe that it is the design philosophy and the size of the implementation, in terms of hardware and software, what makes ALFRED's implementation original. Views, as seen in ADIM [see Chapter 5], have only been implemented in systems that largely exceed the hardware requirements of ALFRED [HSW75, MISTRES]. I also believe that in many of these cases, views have been added as an after thought [MISTRES]. Because of this, the use of views in these systems produces a noticeable degradation in performance.

In addition to the important role that views can play in the definition of logical data bases, as it will be seen in Chapter 5, I believe that they can also contribute to improve the performance of systems of the type of ADIM. They can be used as a convenient way of representing data spread over the network of a distributed data base management system. In ADIM, I sought to incorporate the views and capabilities of ALFRED, at the earliest stages of design. To achieve this, ADIM was implemented as a system of loosely connected multi-processes, which if so wanted could run on a number of different processors, concurrently. A description of this architectural design is given below.

Basically ADIM consists of three types of processing nodes: query generators - G-units, query processors - P-units and one control unit - C-unit. Several G-units and P-units can be connected (using a bus or local area network) to the central C-unit, Fig. 4.1. Normally, a G-unit co-exists with a P-unit in one machine. A brief discussion of the role of these units and their interconnections is the content of this chapter.

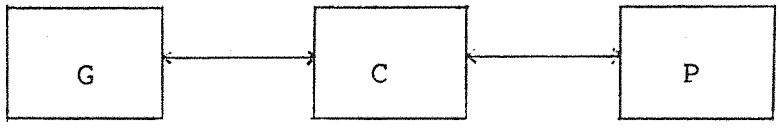


FIG. 4.1

4.2 Query Generators: G-units -

In its most frequent use, a G-unit accepts queries in the ALFRED-U language and prepares reports as produced by the display command. As an alternative method, a query or set of queries can be submitted to a G-unit in the form of a program written in either pure ALFRED-VC or PROLOG with embedded ALFRED-VC/K statements. This latter method can also be used for the preparation of reports.

Queries are normally submitted to ADIM through a G-unit. A query expressed in ALFRED-U or ALFRED-VC has to be passed to the C-unit for decomposition and generation of the equivalent ALFRED-K statements. Queries expressed in ALFRED-U by the G-unit are translated into equivalent ALFRED-VC statements, prior to submission to the C-unit.

Queries in ALFRED-VC form are passed to the C-unit which decomposes them and distributes the processing of the sub-queries over the network of P-units. Queries to P-units are expressed in ALFRED-K. The C-unit returns result relations, normally one, and error conditions, if any.

A G-unit is made up of three modules, as follows:

G-MONITOR -

This module is the outer layer of the system. Queries are entered through the monitor using the ALFRED monitor or a host editor. The ALFRED monitor provides a facility to enter queries at the user's terminal. By using an editor, queries can be written in a file which is given as input to the ALFRED monitor. It is also possible at this stage, to write a pure ALFRED-VC program or a PROLOG program with embedded ALFRED-VC statements. These programs can make calls to the ADIM library.

G-DBMS -

This is a data base that maintains the local schema as its only task. This data base is maintained as a Prolog data base.

G-SCHEMA -

This is a description of what users of an individual G-unit can see of the global system. G-Schema is a data base which keeps information about the relations in the system as seen from this G-unit. Partitions of relations and physical locations are transparent to the G-Schema.

The relations in this data base are represented by Prolog facts.

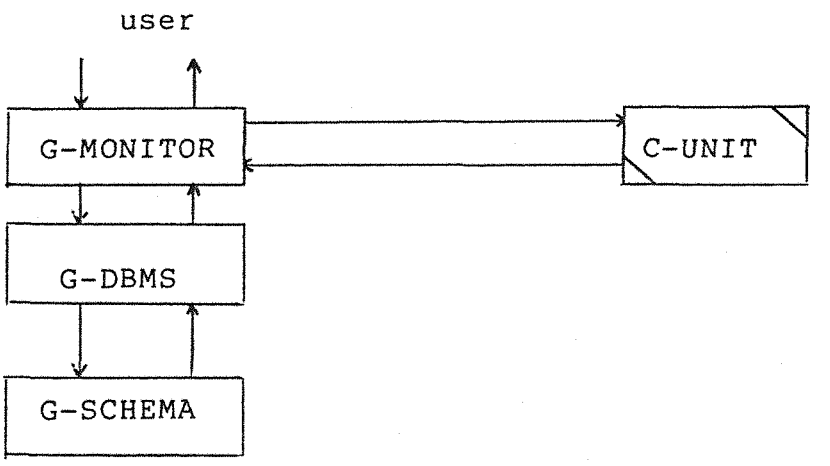


FIG. 4.2

4.3 Query Processors: P-units -

Query processors usually referred to as P-units, are the local processing engines. A P-unit receives queries expressed in ALFRED-K form, processes the queries and returns the result to the calling C-unit. A P-unit only knows about the local relations and therefore, the queries processed by a particular P-unit must be referred to data bases held locally. It should be noticed that P-units not only provide a considerable processing power but also constitute the storage nodes of ADIM.

The modular structure of a P-unit is as follows:

P-PROCESSOR -

This is a processing unit for local queries. This unit is a centralized version of the kernel of ADIM. This local data base management system handles all the data stored in this node. It also includes the schemas for the local data bases which are kept as relations.

P-SCHEMA -

This keeps information on the relations stored in the local data bases.

P-DB -

These are the storage cells of the network. The P-DBs are indeed the local data bases.

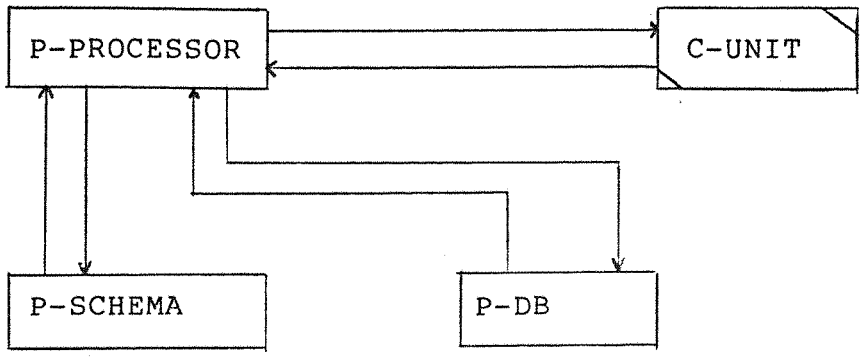


FIG. 4.3

4.4 Control Unit: C-unit -

The C-unit is the centre of the network. Many G-units and P-units may be connected to one C-unit. Every query in ALFRED-VC form is decomposed by the C-unit into a number of local queries. If the query is received by the C-unit in ALFRED-K form, decomposition is not necessary and therefore the C-unit only re-routes the query. Normally, however, queries are received in ALFRED-VC form, they are then decomposed and transformed to ALFRED-K form. The local queries resulting from decomposition are sent by the C-unit to the relevant P-units, which in turn, return an answer. Finally, the C-unit further processes the local answers and a final reply is sent to the G-unit responsible for the original (global) query.

Thus, the C-unit is the centre of control for the whole distributed system. It receives queries, decomposes them into sub-queries, allocates the processing of sub-queries to different P-units, performs joins while composing the reply and ultimately, delivers a relation (or relations or error messages) back to the original source of the query. The different modules to perform all of these tasks are described below:

C-PROCESSOR -

This receives queries in ALFRED-VC form. Once a query has been received, the P-Processor decomposes the query into sub-queries with the support of the C-DBMS, which, in turn holds information about all the relations in the system, i.e. the global schema (see C-Schema below). A stream of sub-queries is passed over to the P-Switch which returns a serial reply of relations and/or error messages. In order to recompose a reply to the original query which has been decomposed into sub-queries, a number of join operations has to be performed. This task is delegated to the C-DBMS by the C-Processor.

C-DBMS -

This is a specialised data base management system. It performs two tasks. Firstly, it supports the C-Processor in decomposing the original query into sub-queries, and secondly, it performs joins on behalf of the C-Processor, so that a composed reply can be obtained from the serial replies produced by the P-Switch. In relation to the first task, it maintains the C-Schema and provides the network administrator with an interface, so that security and integrity constraints can be enforced. This module has been implemented by embedding the kernel of ADIM into Prolog.

C-SCHEMA -

This is the global schema. This is a data base with information about the relations existing in the system. These relations are indeed the users views. Prolog is used to represent the C-SCHEMA.

C-METHA-SCHEMA -

This is the schema describing the data stored in the C-unit, that is the C-SCHEMA. Again, a Prolog data base is used to describe the C-METHA-SCHEMA.

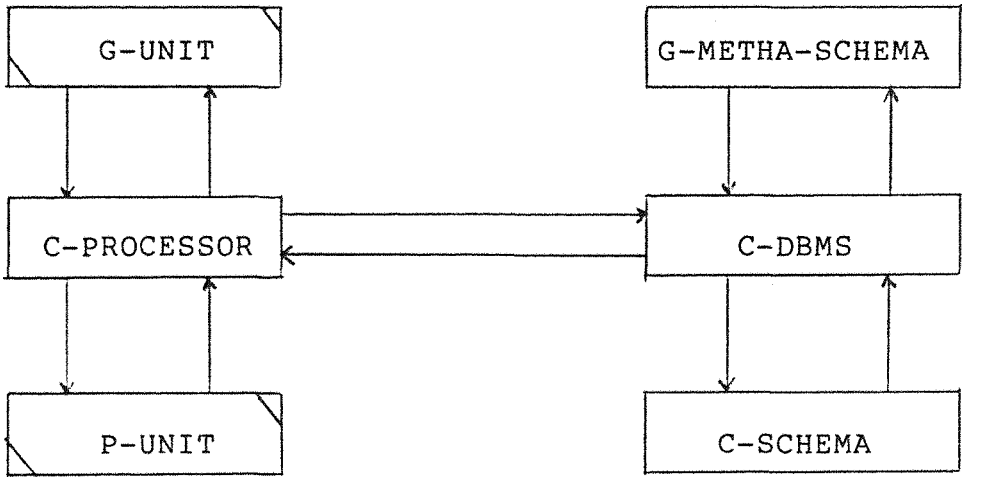


FIG. 4.4

It should be appreciated that this data base is at the centre of control for the whole of ADIM. Perhaps, the most important relations in this data base, are those holding information on the distribution of data. A description of these relations is given below:

1. R_local - maintains information on the relations stored in the different local nodes.

The attributes,

Lname	:	is the local name of the relation.
Node	:	the identifier for the node where the relation is stored.
LogCond	:	is a logical condition attached to the relation, e.g.: "all students in this relation are in the School of Mathematics".
Type	:	it could be public or private.
Owner	:	the owner of the relation.
Cardinality	:	the cardinality of the local relation.

2. R_global - maintains information on relations as seen globally.

The attributes,

Name	:	is the global name of the relation.
Rexpr	:	is the ALFRED-VC expression to form this relation from the local relations.
DAccess, DSave TimeStamp, Semaphore, Owner, Permission	:	are the last access and save up to dates. have the obvious meaning.

3. Node - stores information on the local nodes.

The attributes,

Nodeld	:	is the identifier for the node.
Siteld	:	is the site identifier.
Host	:	is the host computer identifier.
Owner	:	is the owner of the node.

4. Link - maintains information on every communication in the network.

The attributes,

Linkid	:	is the unique identifier for the link.
Nodeld	:	the identifier for the node where this linked is sited.
Direction	:	the direction of the link, i.e.: IN or OUT.

5. Linktype - maintains further information on the link.

The attributes,

Linkid	:	as in the relation Link, it is the identifier for the link.
Protocol	:	the general protocol, e.g.: X25.
Type	:	the specific implementation of the protocol, e.g.: PSS.
Speed	:	speed factor, e.g.: 9600.
CostF	:	the fixed cost of using this link.
CostV	:	cost per unit transmitted.
Class	:	the type of network, e.g.: 1-1, broadcast, etc.
Special	:	special to the link, e.g.: number to dial.

The loading of the information to the different relations of the data base is the task of the data base administrator. For this purpose, a suite of programs to carry out automatic decomposition of views is provided. The following chapter discusses these programs in some detail.

4.5 Segments and Concurrent Processes -

It is my view, that the distinction of three processing units: G, P and C units, is a key element in the provision of full support of segmented logical views of data, in ADIM. At a physical level, the use of separated schemas permits the physical decomposition of relations, thus providing ADIM with an extensive capability for parallelism during query evaluation. I believe this feature to be the most important element towards the development of efficient desk-top information systems. It should also be noticed, that due to hardware limitations, a unique centralized system might be desirable. In this case, all three units could be sited on the one machine and each of the units could be implemented as a separate process. Communications could be established by using intermediate files, or if the host operating system provide them, by pipes.

A more detailed discussion on the use and implementation of views and decomposition techniques is held in the next chapter.

CHAPTER 5

DECOMPOSITION

5.1 Introduction -

Decomposition techniques and methods have several motivations. Among others, they can be used to support different logical views over a common pool of data, to improve the performance of data base management systems and to help to maintain the security and integrity of data bases. This chapter discusses the uses of decomposition techniques in the context of ADIM, and it also describes some theoretical and practical aspects of their implementation in ADIM.

Perhaps, the most obvious usage of decomposition techniques and methods, is to provide support for the co-existence of different views over a common pool of data. This application rests upon logical considerations. To explain the concept of views, let us consider a university data base as example. This data

base consists of the following relations:

```
administration [
                name, address, tutor,
                dateofbirth, startyear, faculty
               ]

mathematics    [
                name, tutor,
                startyear, subject
               ]

physics        [
                name, laboratory, startyear
               ]

computing      [
                name, startyear, tutor,
                laboratory, project
               ]
```

The relation in the data base above, could be interpreted in a number of manners. Let us consider one such interpretation. The central administration of the university keeps personal data about every enrolled student of the university in the relation administration. People in the physics department are only interested in their own students. The same is true in the department of mathematics. In both of these two cases, some additional information is required beyond what the administration can offer. Thus, the need for relation physics and relation mathematics arise. For instance, information about the laboratory used by each student of physics. Since, computing is a group within mathematics, they too would like to keep a copy of some of the data

for mathematics and add to it, information that is specific to the students of computing science. Hence, the existance of the relation computing.

The given interpretation for the example data base, illustrates a case where users needs for information overlap. The data base could indeed be set up as four independently stored files. Alternatively, the four relations could be integrated in such a way that common data is shared, thus avoiding duplicate copies.

This latter alternative immediately solves one problem. Consider a student of mathematics who changes tutor. The situation is recognized within the mathematics department and consequently, the relation mathematics is updated to reflect the change. But since, staff in mathematics have no direct access to the relation administration, no change is made to it. Thus, a problem of integrity within the university data base arises. Information about a particular student is self-contradictory. By holding only one copy of the common data, this problem would have never arisen.

In order to allow users to share common information, and at the same time, to maintain their own associations over the data, separated logical views should be constructed for each group of users. To make this

possible, the information system in use must provide this capability. ADIM's support of views is based upon the use of a number of decomposition techniques.

Efficiency can also be greatly improved by decomposition, since processing of a query can be partitioned into subqueries, each of which could be processed in parallel in a distributed or multiprocessor system. Sotrage use is also improved by sharing a single copy of common data. In a later section on query transformation, I show some techniques which make advantageous use of decomposition. It should also be noticed that the reduction in size for each one of the physically stored relations, means that data flows can be spread more evenly on the system's pathways, thus avoiding major jams in the circulation of data. I believe this last reason, to be a strong argument for the application of decomposition techniques in distributed and/or multiprocessor data base systems. In ADIM's case, this argument is even more relevant given its minimal hardware requirements.

In addition, decomposition techniques can also lead to more secure data bases. In the example, the administration relation could be partitioned into two relations, one relation holding confidential data such as address and date of birth, while the second relation

holds the remaining information.

5.2 Basic Concepts -

Before a more detailed description of decomposition techniques is undertaken, some basic concepts are introduced. Two conceptual operations are defined: D-union and D-join, as well as a number of other related concepts. These definitions follow.

5.2.1 Simple Relation -

A basic relation (or simple relation) in a given data base is a cluster of records representing one partition after decomposition.

5.2.2 D-union -

Relation R is the D-union of relations R' and R", denoted by

$$R = R' + R''$$

if

(1) R' and R" have exactly the same attributes;

(2) R' and R" have the same attributes in their primary key;

(3) the primary key value sets of R' and R", denoted by R'[K':>] (Note: This notation was taken from the book by G.Wiederhold, "Database Design", [WIEDERHOLD]) and R"[K":>] respectively, are mutually exclusive;

then

R = {x/x is either a tuple in R'
or a tuple in R"
}

5.2.3 D-join -

Relation R is the D-join of relations R' and R", denoted by

R = R' * R"

if

(1) R' and R" have the same attributes in their primary key;

(2) R'[K':>] and R"[K":>] hold any of the relationships:

(a) R'[K':>] is a subset of R"[K":>]
(b) R"[K":>] is a subset of R'[K':>]

(3) the attribute sets for R' and R", denoted by R'[A'] and R"[A"] respectively, hold the relationship:

$$(R'[A'] - R'[K':>]) \cap (R"[A"] - R"[K":>]) = []$$

where [] denotes the empty relations, then

R[K:>] denotes the primary key for R and

R = {r/ r' is a tuple in R' and r[A'] = r'
and r" is a tuple in R" and r[A"] = r"
and r[K] = r'[K'] = r"[K"]
}

5.2.4 Compounded Relation -

A compounded relation in a given data base is a relation defined by a decomposition expression. This expression is made up of simple relations, algebra operators in ALFRED, and D-join and D-union.

5.2.5 Characteristic -

The characteristic R<E> of a relation R is the logical expression E such that E evaluates to true for every tuple in R. For example, in our student data base, the characteristic for each relation is:

```
computing < faculty = "mathematics" and
            subject = "computing"
          >

administration < true >

physics < faculty = "physics" >

mathematics < faculty = "mathematics" >
```

The characteristic R<true> is referred to as the universal characteristic.

If confidentiality was to be preserved in some of the information in the relation administration, the decomposition:

$$\text{administration} = P' * P''$$

where

$$P'[\text{name}:>\ \text{address, dateofbirth}]$$

and

$$P''[\text{name}:>\ \text{tutor, startyear, faculty}]$$

could have been established.

5.2.6 Link -

Two views, represented by R' and R" respectively, are said to be <u>linked</u> if

(a) $\exists$ x,y such that $x \in R'\langle E'\rangle$ and $y \in R''\langle E''\rangle$ and x[K':>] = y[K":>]

(b) $(R'[A'] - R'[K':>]) \cap (R''[A''] - R''[K'':>]) \neq []$

5.3 Decomposition Procedure -

The interpretation of the world albeit a small part of it, is a human activity. A comprehensive treatment of the design of models to represent reality escapes the boundaries of this work. Nevertheless some practical help is useful. Thus, ADIM provides data base designers with a number of tools to aid the design of data bases. It should be reminded though, that the ultimate responsibility rests upon the people designing the data bases.

As expressed in the previous section, D-union and D-join only exist as conceptual tools of analysis. The same applies to the procedure for decomposition presented below. Since, ADIM's retrieval performance is highly dependent on the physical size of relations, rather than in the number of relations involved in a query, I have devised a decomposition procedure such that physically large relations can be represented by compounded relations made up of several small basic relations.

Let us begin with a matter of notation. The partition i of relation R is denoted by R[i]. Then, the decomposition procedure for a given set of views over a common pool of data, is:

Step (1). Create 3 lists:

PARTITIONS, denoted by P.
Initially, it holds all the views in the data base. For each view, an entry exists in this list. An entry has three fields:

(a) a unique name for the partition, say R';
(b) the attribute set for the partition, i.e. R'[A'];
(c) the characteristic for the partition, i.e. R'<E'>.

SCRATH, denoted by S.
Initially, it holds all the names of partitions, i.e. names in field (a) of P.

EXPRESSIONS, denoted by E.
Each entry in this list has the form:

<view> = <decomposition expression>

where, <view> is the name of the view, and <decomposition expression> is the expression denoting this view, i.e. a compounded relation. Initially, the list E holds the unique names in field (a) of P, in both sides of the '=' symbol.

Step (2). Apply the procedure below. The element i of list P is denoted by P[i]. The same applies to lists S and E. The last element in a list is denoted by LAST. In C-like notation, the procedure is again:

```
for (i=1; i<LAST; i++)
    for (j=1; j<=LAST; j++)
        if (S[i] is linked to S[j])
        {
         partition (S[i],S[j]); /*given
                                  below*/
         enter resulting partitions
               at end of P;
         delete from P the entries for
               S[i] and S[j];
         delete S[i] and S[j] from S;
         in E, replace all occurrences
               of S[i] and S[j] on
               the rhs of expressions,
               by their equivalent
               expression, using the new
               partitions and the D-join
               and D-union operators;

         goto again;
        }
```

Step (3). Stop. The list E holds the relevant expressions.

The procedure partition (R', R") to partition the views/relations R' and R" completes the general decomposition procedure. The details of it, follow.

Step (1). Rearrange the order of attributes in R' and R" so that they only intersect in one common area. Notice that the tuples have not been input yet, thus, one could imagine a reordering of tuples in both relation such that the intersection of R' with R" only occurs in one common area. In pictures,

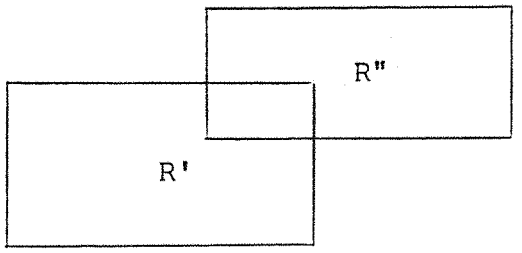


Fig. 5.1

Step (2). A. Get $R[A] = R'[A'] \cup R''[A'']$

B. Divide R[A] into three sets:

(a) $I[A] = R'[A'] \cap R''[A'']$,

notice that

$R[K:>] = R'[K':>] = R''[K'':>]$

is a subset of

$I[K:>]$;

(b) $R'''[A'''] = R'[A'] - I[A]$

(c) $R''''[A''''] = R''[A''] - I[A]$

In pictures,

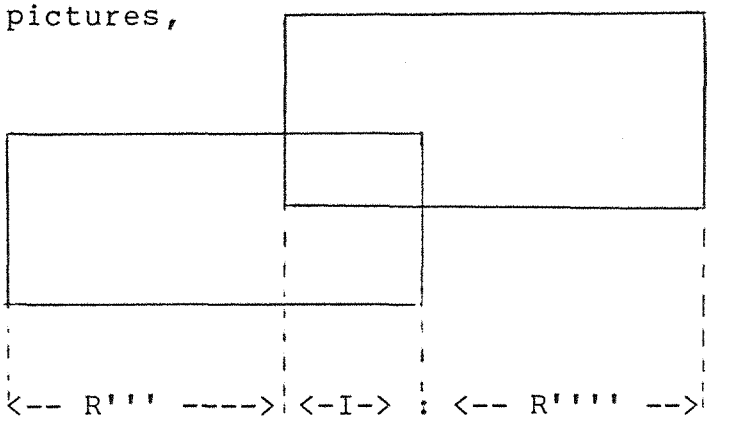


Fig. 5.2

Step (3). This partition of R' and R" determines three possible relations: R''', R'''' and I. Now, let us consider the relation I. If R'<E'> was the characteristic of R' and R"<E"> was the characteristic of R", we further divide I, horizontally, by decomposing I into the tree relations defined by the characteristics R'<E'> and R"<E">. Thus, we have

I = I' + I" + I"

with characteristics for I', I" and I'''.

I'<E' and not E">
I"<E" and not E'>
I'''<E' and E">

and attributes:

I[A] = I'[A] = I"[A] = I'''[A].

It should be noticed, the importance of functional dependencies [ULLMAN, VEMAD], in the above procedure. Their identification by the data base designer, can produce results even when the attributes involved in the characteristics of R' and R" are not the same. To explain this, consider our students data base, again. Let us imagine a super-relation covering the whole data base. Whenever the attribute subject takes the value "computing", the attribute faculty must necessarily take the value "mathematics". This dependency allows us to partition the mathematics relation into two relations: R' and R", with characteristics: R'<subject = "computing"> and R"<not (subject = "computing")>, respectively.

Step (4). Finally, the equivalent expressions for R' and R" can be constructed. In the general case:

R' = R''' * (I' + I''')
R" = R'''' * (I" + I''')

and, in the special cases:

For R', if

(a) I' = [] then R' = R''' * I'''
(b) R''' = [] then R' = I' + I'''
(c) R''' = [] and I' = [] then R' = I'''

Similarly, for R", if

(a) I" = [] then R" = R'''' * I'''
(b) R'''' = [] then R" = I" + I'''
(c) R'''' = [] and I" = [] then R" = I'''

The application of the decomposition procedure to our example data base, after 44 iterations in Step (2) produces the following list of expressions E:

```
administration = ((R1Ø+R11)+(R7+R6)*R8)*R3
mathematics    = R1*(R1Ø+R11)
physics        = R5*R6
computing      = R9*R1Ø
```

where,

R1 [name, subject]
R1 < faculty = "mathematics" >

R3 [name, dateofbirth, address, faculty]
R3 < true >

R5 [name, laboratory]
R5 < faculty = "physics" >

R6 [name, startyear]
R6 < faculty = "physics" >

```
R7 [name, startyear]
       R7 < not (faculty = "mathematics" and
                 faculty = "physics") >

R8 [name, tutor]
       R8 < not (faculty = "mathematics") >

R9 [name, laboratory, project]
       R9 < faculty = "mathematics" and
            subject = "computing" >

R1Ø [name, startyear, tutor]
       R1Ø < faculty = "mathematics" and
             subject = "computing" >

R11 [name, startyear, tutor]
       R11 < faculty = "mathematics" and not
             (subject = "computing") >
```

To obtain R9, R1Ø and R11, the dependency:

subject --> faculty was assumed.

5.4 Design Tools -

The use of functional dependencies [ULLMAN] in the decomposition procedure in the previous section, highlights the need for some analysis tools to help the design of data bases. ADIM provides two such design tools: TC - a program to aid in the identification of functional dependencies and AT - a program to help in the analysis of entities [VETMAD] and their relationships. A brief description of these programs follow.

5.4.1 TC - Functional Dependencies -

It is because of the relative importance that functional dependencies have in the decomposition procedure, that a program to aid in their identification, is included in ADIM. Before we explain this program, some basic concepts have to be defined.

A functional dependency fd is defined as follows:

Attribute B is functionally dependent on set of attributes A = $\{A_1, A_2, \ldots, A_n\}$, denoted by

$fd\ (A_1, A_2, \ldots, A_n) = B$

if the value v[B] in any tuple v, is always fully determined by the set of values

$\{v[A_1], v[A_2], \ldots, v[A_n]\}$

Armstrong's axiom on transitivity can be stated as:

If $fd(A_1, A_2, \ldots, A_n) = B_1, B_2, \ldots, B_m$
and $fd(B_1^1, B_2^2, \ldots, B_n^m) = C_1^1, C_2^2, \ldots, C_1^1$
then $fd(A_1^1, A_2^2, \ldots, A_n^m) = C_1^1, C_2^2, \ldots, C_1^1$

Now, we are in a position to discuss the program TC. The designer of the data base identifies some of the functional dependencies in the data. Then, s/he feeds these dependencies to TC, which by recursive application

of the transitivity axiom generates the transitive closure (TC) of the input set.

The program and an initial set of fds is presented below. The set of fd's:

```
fd ([name], [address]).
fd ([name], [spouse]).
fd ([spouse], [children]).
```

illustrates the way in which the program is initialized for a starting set of fd's.

```
/*
        TEST DATA BASE
*/

fd([name], [address]).
fd([name], [spouse]).
fd([spouse], [children]).

/* transitivity */
trans( AttL1, AttL2 ) :-
        fd( AttL1, X ), fd( X, AttL2 ).

/*
        TRANSITIVE CLOSURE
*/

tc([]).
tc( [ X | Y ] ) :-
        /* generator */
        (
                ( fd( X, Z )
                  ;  /* or */
                  ( not var( X ),
                    trans( X, Z ),
                    not fd( X, Z ),
                    not var( Z ),
                    asserta( fd( X, Z ) ),
                    /* mark new fd added */
                    asserta( newfd( mark ) )   /* only once */
                  )
                ),
                fail
        )
        ;
        tc( Y ).


/*
        GENERATE TC
*/

/* general case */
gentc( [ X | Y ] ) :-
        /* generate all by recursion */
        tc( [ X | Y ] ),          /* level 1 */
        /* check if any new fd added */
        newfd( mark ),
        retract( newfd( mark ) ),
        /* recursion one level down */
        gentc( [ X | Y ] ).

/* boundary condition */
gentc( _ ) :-
        /* no new fd added */
        listing( fd ).
```

The AXIOMS section of the program, defines the rule trans(...) for the transitivity axiom. In a more comprehensive version of this program, other axioms have also been included. Axioms to test reflexivity, augmentation, union and decomposition [ULLMAN] are included in this more sophisticated version of TC. As it can be appreciated in the version of TC here presented, the definition of Prolog rules for the axioms is quite straightforward.

In this version of the program TC, the second definition of tc(...) acts as a generator of possible fd's and tester for them. Once a satisfactory fd is found, it is added to the other fd's by asserta(...).

The rule gentc(...) activates the previous rule, tc(...), in a recursive manner.

Finally, the lists of fd's produced by TC is examined by the data base designer, who chooses a suitable set of non-redundant fd's for use with the decomposition procedure reported in the previous section.

5.4.2 AT - Analyst Tool -

The software tool here discussed, denominated AT - Analyst Tool, helps the analyst/data base administrator in the determination of elementary relations [VETMAD] to construct data bases. The program is based on the identification of entities and their relationships.

Data bases in AT are formed by the application of mechanical rules to entities and relationships discovered by the analyst within the organization being modelled. AT does not replace the analysis process; this is still done by the analyst by means of interviews and consultation of the relevant documents in the organization.

Once entities and relationships are identified by the analyst, AT queries him/her and determines the elementary relations necessary to represent all the different conceptual views that users may have of their organization/activity. Queries to the data base administrator/analyst try to determine the following:

i) entities and their names (unique);
ii) domains and their names (unique);
iii) primary keys for each type of entity; and
iv) relationships among entities and their names (unique).

Once this information is provided to AT by the analyst, AT proceeds to:

i) form virtual relations for the entities; and
ii) form virtual relations for the relationships.

Then, from these virtual relations, AT seeks to discover sets of elementary relations. These are obtained by asking the analyst for the functional dependencies that have been identified by him/her in the analysis process. The program TC which was presented earlier on, aids the analyst in this task.

The elementary relations of this stage are integrated into ADIM's data bases by means of the simple techniques described in section 5.6. Obviously, at this stage minimal covers could also be determined. The future development of a program to do this, depends on the results of further research on the relationship between a minimal cover determined purely by functional dependencies and the decomposition of views advocated earlier on in this chapter.

5.5 Retrieval Tactics -

Let us now explore the potential for efficiency improvements of decomposition. By using our students data base and ADIM's algebra query language an example can be given.

Consider the question:

> "list the name and date of birth of students in the faculty of mathematics"

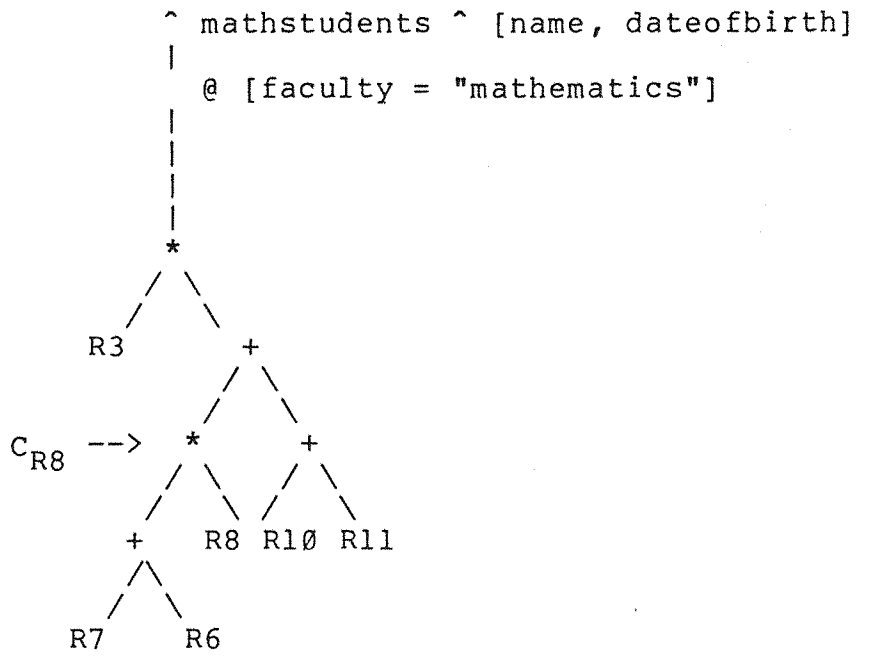
expressed in ALFRED-U, as

```
RETRIEVE administration WHEN [faculty =
                                "mathematics"]

INTO mathstudents ^ [name, dateofbirth]?

DISPLAY mathstudents?
```

the equivalent tree for this expression is

```
mathstudents ^ [name, dateofbirth]
  |
    @ [faculty = "mathematics"]
  |
  |
administration
```

Using the decomposition of relation administration, the tree becomes



By defining

(a) R + []=R
(b) R * []=[]

and, denoting the retireval condition by Q, we have

Q=[faculty="mathematics"]

and, since

(R8<not(faculty="mathematics")> and Q)=[]

then the whole of the arrowed subtree can be eliminated from the evaluation of Q.

Similarly, deletions, insertions and updates can be handled.

Consider another example, where efficiency improvements are introduced by the application of query transformation techniques [PECHERER, PALERMO]. Suppose

that in the students data base, we were concerned with the age (date of birth) of those students reading computer science.

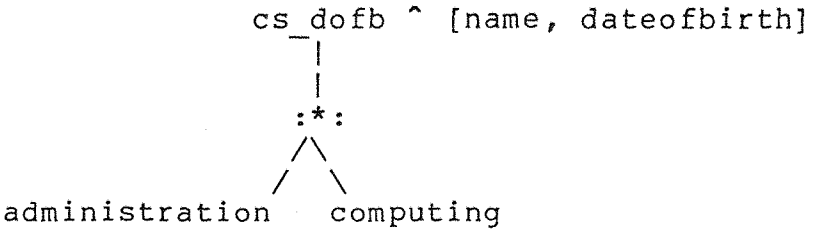
In plain english:

> "list the date of birth of students reading computer studies"

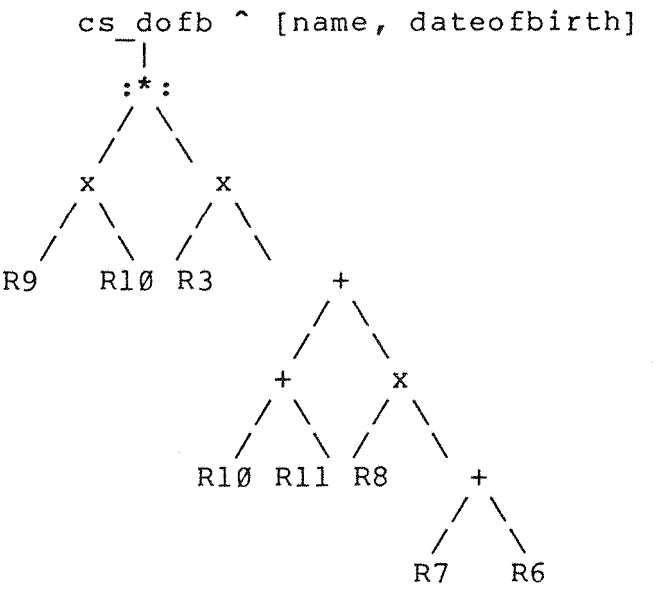
In ALFRED-U:

```
RETRIEVE administration :*: computing
INTO cs_dofb ^ [name, dateofbirth]?
```

The corresponding evaluation tree for the query is:



Replacing the relations administration and computing by their respective decomposition expressions, the tree becomes



Clearly, in order to evaluate the decomposed tree, it is not necessary to wait for the expression "R9*R1Ø" to be evaluated. In fact, it is more efficient to modify the tree, bringing the evaluation of the projection ^[name, dateofbirth] down the tree, thus reducing the amount of data required to pass between the different nodes of the tree. Also, notice that once the projection is lowered in the tree, the evaluation of R9^[name, dateofbirth]becomes R9^[name] * R1Ø^[name] since the attribute dateofbirth is in neither R9 nor R1Ø. Furthermore, this expression can be replaced, after R9 (or R1Ø) has been selected because of its (small) cardinality, by the expression

R9 ^ [name]

if cardinality of R9 is less or equal
than the cardinality of R1Ø

The correctness of the last step is due to condition (2) in the definition of d-join. In other words, given relations R' and R" with the common primary key K

$$R'[K:>] * R"[K:>] = R'[K:>]$$

whenever the value set for K in R' is a subset of the value set for K in R". A proof of this assertion is trivial.

From the example, it emerges clearly that substantial benefits can be obtained by query transformation and decomposition. To determine what relationships hold among the operators, and which relationships to apply in a particular situation are not by any account simple problems. Relationships among the algebra operators of the query language have been established in several cases, and their conditions for optimal use have been specified as well [PALERMO, PECHERER]. The integrated study of decomposition operators and query algebra operators need some further work. ADIM's approach is of a pragmatic nature, where complexity analysis define rules of transformation of a general standing, ADIM uses them; but in the particular cases where costs can be estimated with a certain degree of accuracy, specific strategies are adopted.

An example of this is often provided by the join operation. Any analysis based on its complexity would suggest that evaluation of joins should be postponed for as long as possible, but if it was known that the two relations involved in the join have a small cardinality, one could in some cases favour an early evaluation.

Before we leave the discussion on retrieval tactics and perhaps, as a suitable introduction to the next section on implementation, it should be noticed that the definitions of D-union and D-join, in fact, correspond to special cases of union and natural join, as defined in [CODD7Ø]. This contradicts some researchers who have suggested decomposition operators which do not have a counterpart in Codd's algebra [SKCWHC, CODD7Ø]. ADIM's set of operators allows the translation phase of the optimizer to embed compounded relations into a query expressed in algebraic form, and then translate the expanded query into a simpler and/or more efficient query. It is worth noticing the considerable results obtained by many researchers using translation techniques to optimize queries in algebraic languages [PALERMO, PECHERER, SAGIV]; where these languages mirror closely Codd's algebra. Most of these results could not be fully useable by ADIM, if an uncompatible set of decomposition operators was chosen. As a limiting factor, although decomposition techniques can handle volatile data

advantageously, stable user's views have to be assumed, in order to avoid very expensive re-construction of data bases. On the other hand, from an implementation point of view, since D-join and D-union are special cases of union and natural join, no special software modules are necessary in the implementation of the ADIM system.

5.6 Implementation -

Once the data base administrator has decided on a suitable set of basic relations, views and characteristics, a data base has to be set up.

Basic relations, views and characteristics are established in the data base as Prolog facts. An example of this is given below:

```
/*DATA BASE : demo*/

/*relations*/
relation( employee).
relation( students).
     .
     .
     .

/*characteristics*/
characteristic(students,[dept = "mathematics]).
     .
     .
     .

/*views*/
view(lowpaid, @[salary<10000]).
     .
     .
     .
```

A basic relation is defined by the relationship relation. The name of the basic relation appears as the only object of the Prolog fact. For example,

```
relation(section A_staff).
```

defines the basic relation section A_staff. Basic relations are physically stored in secondary memory, e.g. disc, and can only be accessed by ALFRED-K requests.

The characteristic of a basic relation is defined by a restriction condition and it is expressed on those terms. Thus a characteristic is represented by the Prolog fact characteristic(...) which has two objects: the name of the relation and the restriction condition.

For example,

```
characteristic(section A_staff,[section = "A"]).
```

establishes that the characteristic of the basic relation section A_staff is [section = "A"]. At present, the values for the attribute section are still stored, but obviously they are redundant information. Future work on ADIM should seek to correct this.

Basic relations without a characteristic are assumed to have the universal characteristic, i.e [true].

Again, views are also represented as Prolog facts. The view relationship has two objects: the name of the view and the algebraic expression associated with the view's name. For example,

```
view(dept1_staff, sectionA_staff :+: sectionB_staff).
```

defines the view dept1_staff as the union of the basic relations sectionA_staff and sectionB_staff.

In fact, the expression defining a view is not restricted to the use of basic relations. Views can also appear in an expression defining a new view. For instance,

```
view(staff, dept1_staff :+: dept2_staff).
```

defines the view staff as the union of the views dept1_staff and dept2_staff.

It should also be noticed that by allowing the assignment of a characteristic to a view, ADIM allows general characteristics to be propagated to many basic relations. For instance,

```
characteristic(staff, [site = "Southampton"]).
```

propagates all the way down the tree the characteristic [site = "Southampton"]. This characteristic is thus shared by several basic relations. In this example, basic relations sectionA_staff and sectionB_staff inherit the characteristic [site = "Southampton"].

5.6.1 Generation of ALFRED-K queries -

As already mentioned, ALFRED-U expressions are mapped into ALFRED-VC by the G-monitor in the Query Generator (G-unit, Chapter 4). In turn, ALFRED-VC expressions are further processed by the C-unit, and equivalent expressions in ALFRED-K are generated. ALFRED-K expressions only admit basic relations, i.e. relations

which are physically stored. In the process to convert ALFRED-VC expressions into ALFRED-K expressions, characteristics are added to views and basic relations, views are expanded to expressions made up of basic relations only, and finally, these last expressions are optimized for evaluation. A brief discussion of the query evaluation process follows in this section.

ALFRED: U to VC -

The translation of ALFRED-U expressions to ALFRED-VC expressions is accomplished by an interpreter program. This interpreter is written in Prolog. The program is divided into two sections.

The first of these sections reads ALFRED-U sentences and converts each one of them into a list of Prolog atoms. The second section transforms lists of atoms into ALFRED-VC expressions.

The implementation of the first section is based upon a similar program presented by W. Clocksin and W. Mellish in their book "Programming in Prolog" (pp. 87-88) [CLOMEL]. Obviously, some modifications were necessary to handle the peculiarities of ALFRED syntax.

The implementation of the second section of the program is rather simple and it will not be described here. Nevertheless, a short description of its function is given below.

Basically, the second section takes the list of atoms generated by the first section and converts it into an expression in ALFRED-K form. In the majority of cases, the list of atoms remains unaltered. For instance, the list [display, employee] derived from the ALFRED command:

DISPLAY employee?

will still be the list [display, employee] when passed to the ALFRED:VC to K translator.

However, more complex commands give rise to some interesting problems. In particular, queries could often lead to expressions that would be very inefficient to evaluate directly. Thus, in order to improve the evaluation time of these queries, they are transformed into equivalent queries, which can then be evaluated more efficiently. Cases of this sort are not always due to poorly formulated queries. They also arise because of the incorporation of views and characteristics into the query. Views and characteristics are added to the query by the ALFRED:VC to K translator. Also, steps for a more

efficient evaluation of the query are taken by this translator (VC to K). Nevertheless, the ALFRED-U to VC translator 'optimize' the query, by altering the syntax within the list of atoms. This new syntax, while still readable and understandable by ordinary users, is more amenable to further manipulation than the one used at the end of the first section.

ALFRED: VC to K -

Queries in ALFRED-VC are translated into ALFRED-K equivalent queries, in three stages. This is done by an interpreter which is also written in Prolog. The first part of this program defines the syntax, priority and associativity rules for operators. The following extract from this part defines the operators restriction, union, join and projection, respectively.

```
? - op(7, xfy, @).          /*restriction*/
? - op(9, yfx, :+:).        /*union*?
? - op(1Ø, yfx, :+:).       /*join*/
? - op(8, xfy, ^).          /*project*/
```

The three stages for this interpreter are: characteristic handler, view explosion and optimizer. The Prolog rule map(E,F) transforms the ALFRED-VC expression E into the ALFRED-K expression F.

```
/*
    map(E,F):-
          maps expression E into the fully
          decomposed, optimized expression
          F (in clausal form).
*/

map(E,F):-
          char(E,E1),     /*add characteristics*/
          expl(E1,E2),    /*explode views*/
          simp(E2,F).     /*optimize*/
```

As it can be seen above, the rule map(E,F) sequentially activates the tree stages.

Characteristics are added to basic relations and views by the following rules:

```
/*add characteristics*/

char(E,F):-          /*form the restriction
                       expression*/
         characteristic(E,C),
         F = .. [@,E,C].

char(E,F):-          /*recursive propagation*/
         E = .. [Op,Lexp,Rexp],
         char(Lexp,Xexp),      /*left hand side*/
         char(Rexp,Yexp),      /*r.h.s.*/
         F = .. [Op,Xexp,Yexp].

char(E,E).           /*catch-all*/
```

Three similar rules govern the expansion of views:

```
/*
    expl(E,F):-
                expands views in expression E
                to expression F which has basic
                relations only.
*/

expl(E,E):-   /*catch basic rels, attributes and
                comds*/
          basic(E),!.  /*test E is primitive object*/

expl(E,F):-   /*explodes views*/
          view(E,El),  /*is E a view?  expand it*/
          expl(El,F).  /*explode El*/

expl(E,F):-   /*recursive explosion of views*/
          E = .. [Op, Lexp, Rexp],
          expl(Lexp, Xexp),
          expl(Rexp, Yexp),
          F = .. [Op, Xexp, Yexp].
```

The first of the rules is the catch-all rule. It has a few exceptions. It takes account of basic relations, lists of attributes and lists of conditions. The second rule, once it finds a view, expands it in case there are more views hidden in a tree of views. The last rule, propagates the expansion along the expression E in a recursive manner.

Optimization or rather efficiency improvements are governed by the rule simpl(E,F). This rule transforms expression E into expression F. Expression F is equivalent to E but in most circumstances it will be evaluated in a time that is significantly faster than the time it would take to evaluate E. The rules for this transformation are:

```
/*
    simplify relational expressions
*/

simpl(E,E):-      /*catch-all basic atoms*/
          basic(E),!.

simpl(E,F):-
          E = .. [Op, Lexp, Rexp],
          simpl (Lexp, Xexp),
          simpl (Rexp, Yexp),
          s (Op, Xexp, Yexp, F).
```

The first occurence of rule simpl(E,E) defines the stop condition for the recursion in the second definition of simpl(E,F). This second occurence of rule simpl(E,F) propagates the optimization process, recursively. The clause s(Op, Xexp, Yexp, F) in this rule, is satisfied if a known rule of optimization exists for the operator Op in the context of expressions Xexp and Yexp. Thus, the rule:

```
/*projection associative case*/
s(^,X,Y,Z):-
          is_list(X),
          is_list(Y),
          intersection(X,Y,Z1),
          set(Z1,Z).   /*eliminate duplicates from
                         list Z1*/
```

transforms two adjacent projections on one relation into one projection on the same relation.

For example, let us consider the following expression to optimize:

```
? - simpl(employee^[name, address]^[name, address,
          salary,X).
```

The rule simpl(E,E) will break the expression into its basic parts: the relation employee, the list of attributes [name, address] and the second list of attributes [name, address, salary]. Since the associativity of ^ was defined to be right to left, by the definition

```
? - op(8, xfy, ^).      /*project*/
```

the expression will be interpreted to be

```
employee^([name, address]^[name, address, salary])
```

and hence, when an attempt is made to satisfy the second definition of simpl(E,F), Lexp is instantiated to employee and Rexp is instantiated to [name, address]^[name, address, salary]. Thus, two recursive invocations of simpl(E,F) are made:

```
simpl(employee, _25)
simpl([name, address]^[name, address, salary], _26)
```

The first of these invocations, simpl(employee, _25) instantiates _25 to employee, since employee is satisfied

by basic(employee). It is the second invocation which merges the two attributes lists into one. The merger is done by the associativity rule for projections

```
s(^, [name, address], [name, address, salary],
 _101)
```

which instantiates _101 to [name, address], thus instantiating the variable X to

```
employee^[name, address]
```

In this manner then, the expression employee^[name, address]^[name, address, salary] is simplified to the expression employee^[name, address] which involves only one projection instead of the original two.

Notice that if in the above example, two disjoint sets of attributes were given, the simplification of the expression would reduce the two lists of attributes to the empty set. The projection of a relation on an empty set of attributes has been defined so as to produce the empty relation by the rule:

```
/*empty list of attributes => empty relation*/
s(^, X, [], []).
```

and hence, when this case arises there is no need to inspect relation X.

All of the rules of optimization discussed in section 5.5, have been incorporated into the translator in a similar manner. More importantly, new optimization rules can easily be added to the translator here described. Similarly, most of the rules presented by Palermo [PALERMO] have been incorporated. The same applies to the set of optimization rules discussed by Pecherer [PECHERER]. It should be noticed though that Pecherer's set of rules is a superset of Palermo's set of rules. Some further details about the incorporation of these rules are given later on.

Perhaps, what is new in my set of rules is the treatment of empty lists of attributes, empty relations, unconditional true and unconditional false.

Although, one would not normally expect users of ADIM to submit queries involving empty relations, they might appear in query expressions once characteristics are added to queries and views are expanded into expressions made up of basic relations only. Take for example, the view staff below, on which a query sub-expression is based. Let us assume that the sub-expression being evaluated is

... staff@[salary>10000] ...

and that the view staff has been defined by the data base administrator by

view (staff, employee @ [salary<10000]).

then, the evaluation of this expression will produce the empty relation.

Similar things can happen during the evaluation of projections. We might end up with a sub-expression containing an empty list of attributes on which to project. Such a case presents itself when given a relation, two or more projections are attempted on this relation, using disjoint sets of attributes for the respective projections. For example, the two projections on staff, below

staff^[name, salary]^[code, dept]

are equivalent to

staff^[]

ADIM evaluates this expression to the empty relation. This is obviously, one possible way of interpreting the expression staff^[]. Other people might interpret it

differently, for instance, they could equal it to the relation staff, itself. I decided to use the current interpretation purely on the grounds of consistency, which I expect will become apparent later on in this section.

Also, views and characteristics can produce interesting results. They often turn a condition into a certainty. For example, take the view

```
view (sectionB, staff@[section = "B"])
```

and the sub-expression of a query

```
... sectionB @[section = "B"] ...
```

which is certainly true for every tuple in section "B". Alternatively, take the view

```
view (lowpaid, staff@[salary<7500]).
```

and the sub-expression of a query

```
... lowpaid @[salary>7500] ...
```

which is false for every tuple in lowpaid.

Empty relations and empty lists of attributes or conditions are represented within the ALFRED-VC to ALFRED-K translator by an empty list, []. Unconditional false is represented by the singleton [false], and likewise, unconditional true by [true].

To know that a particular expression or sub-expression evaluates to the empty relation can be used to our advantage. The same applies to lists of attributes and/or conditions evaluating to [true] or [false]. To illustrate this point, take the following user's query:

```
RETRIEVE lowpaid WHERE [salary > 75ØØ]
                 INTO notsobad?
```

If lowpaid was defined as the view:

```
view(lowpaid, staff@[salary<75ØØ]).
```

the ALFRED:VC to K translator would transform the query into the following sequence of equivalent expressions:

```
      .
      .
      .
lowpaid @[salary>7500]
      .
      .
      .
staff @[salary<7500] @ [salary>7500]
      .
      .
      .
staff @[salary<7500 and salary>7500]
      .
      .
      .
staff @[false]
      .
      .
      .
     []                        /*the empty relation*/
```

of which the last one clearly establishes the futility of calling the ALFRED-K processor (P-unit) for this query since at this stage, we already know that the final result is the empty relation.

In the case of restrictions, in order to compare the different conditions of the restriction among themselves, a canonical representation for the condition(s) is necessary. For instance, if we were given the condition [salary>7000 and not (salary>7000)], we would like to match the first occurence of salary>7000 with its negation later on, not (salary>7000). This would allow us to transform the original expression into [false], and consequently, to deduce that regardless of the relation to which the restriction was applied to, the final result

is the empty relation.

Because of the reasons given above, restriction conditions are transformed within the ALFRED:VC to K translator into clausal form. The part of the translator that does this transformation is based on a similar program, which is described in detail in Appendix B of the book by W.F. Clocksin and C.S. Mellish, "Programming in Prolog" [CLOMEL]. Thus, by using this module, restriction conditions received in ALFRED-VC form, from G-units, are transformed into a list of clauses in conjunctive normal form.

A simple example of transformation to clausal form is the mapping of the condition [salary>1ØØ] into the list of clauses [:([salary>1ØØ],[])]. This list has only one clause, :([salary>1ØØ],[]), which in turn, is made up of two lists of disjunctions; [salary>1ØØ] and []. The first of the lists holds the conditions (disjunctions) which are not negated, in this case, the only condition of the restriction, salary>1ØØ. The second list holds the conditions which are negated, in this case, none.

To further illustrate the transformation of ALFRED-VC restriction conditions into clausal form, a list of restriction conditions together with their equivalent clausal form are presented below. These transformations,

I believe, are self-explanatory:

1. staff @[salary>100 and ~(dept="B") and sex="M"]

```
staff @[:([salary>100],[]),
        :([],[dept="B"]),
        :([sex="M"],[])
       ]
```

2. staff @[salary=1000 or dept="B"]

```
staff @[:([salary=1000, dept="B"])]
```

3. staff @[salary>200 or ~(dept="B")]

```
staff @[:([salary>200], [dept="B"])]
```

4. staff @[salary>200 or (dept="B" and sex="M")]

```
staff @[:([salary>200, dept="B"], []),
        :([salary>200, sex="M"], [])
       ]
```

Once restriction conditions have been transformed to their equivalent clausal form, their optimization becomes simpler. For instance, the appearance of a condition p in both, the list of not negated conditions and the list of negated conditions, implies that p is being or-ed with its negation, i.e. p or not p, which is true in all cases. In this case, once this situation is identified the particular clause can be replaced by [true]. This precise instance of clausal optimization is performed within the interpreter by

```
/*test for contradiction*/
contrary (:(A,B),[:(A1,B1)|_]):-
          equivalent (A,A1),
          equivalent (B,B1).
contrary (:(A,B), [_|Cls]):-
          contrary (:(A,B), Cls).
```

The second definition of contrary (...) above, recursively searches for the negation of conditions in A, while the first definition performs the actual tests. The Prolog clause equivalent(...) has been defined elsewhere, and it tests the equivalence of two sets.

Once a condition becomes [true] or [false] further performance improvements for the whole restriction expression can be obtained by the application of the following rules:

```
/*false & X => false*/
optcls(X,[false]):-
          member(X,[false]).

/*true & X => X*/
optcls(X,Y):-
          delete([true],X,Y).
```

In the definition of rule optcls(...), if [false] is found in the canonical expression X, the whole expression is transformed into [false]. This is trivially derived from p &...& [false] &...& r => [false]. Similarly, if the canonical expression X has not been reduced to [false] by the above rule, all occurences of [true] are removed from X by the second definition of the optcls(...) rule. This, in turn, is derived from p & [true] => p.

The transformation of restriction conditions to

clausal form allows ADIM to optimize the evaluation of restrictions, in general. More importantly though, further possibilities of optimizing whole relational expressions arise. For instance, after the explosion of views into their corresponding basic relations expressions, we could have the query:

staff @[salary>100 and ~(salary>100)]^[name]

which, by application of the optimization rules for restriction could be reduced to:

staff @ [false] ^ name

This expression, in turn, could be transformed to the empty relation projected on attribute name, i.e.

[]^[name]

and, then the empty relation, i.e.

[]

The transformation of staff @[false]^[name] into []^[name] is based upon the rule for restriction which states that any relation restricted on the condition [false] evaluates to the empty relation, []. The

transformation from []^[name] to [] is based upon the rule for projection which states that the empty relation projected on any list of attributes evaluates to the empty relation. This later rule is stated in Prolog as the fact:

```
s(^, [], _, []).
```

Other rules for other operators can be stated in a very similar manner. Below, some of these rules have been selected for discussion. Because of the relevance to ADIM's implementation, I have focused the discussion on those rules involving empty relations, empty lists of attributes or conditions, [false] and [true].

Rules for union:

```
s(:+:, X, [], X).
s(:+:, [], X, X).
```

these two rules state that the union of a relation X and the empty relation evaluates to relation X.

For join:

```
s(:*:, _, [], []).
s(:*:, [], _, []).
```

these two rules state that the join of any relation and the empty relation evaluates to the empty relation.

For difference:

```
s(:-:, [], _, []).
```

states that the empty relation difference any relation always produces the empty relation as result; and

```
s(:-:, X, [], X).
```

states that the relation X difference the empty relation evaluates to X.

For intersection:

```
s(:.:, [], _, []).
s(:.:, _, [], []).
```

state that the intersection of the empty relation with any relation, including the empty relation itself, evaluates to the empty relation.

For restriction, before we can apply the relational optimization rules, we need to transform the restriction condition into its equivalent clausal form, which in

turn, can be optimized. The transformation to clausal form and the optimization of it, and the subsequent optimization of the relational expression is performed by:

```
s(@, X, Y, Z):-
        clauseform(X, X1),
        clauseform(Y, Y1),
        srestr(@, X1, Y1, Z).
```

The transformation into clausal form is performed by clauseform(...) which in turn, invokes rule optcl(...) to optimize the clauses. The rule clauseform(...), as implemented in ADIM, is identical to the one presented by W. Clockin and C. Mellish, except by the call to optcls(...) and in some minor syntax details which are specific to ADIM.

Thus, once the restriction condition has been clausified and then optimized by clauseform(...) and optcls(...), respectively, the relational expression of which the restriction is a sub-expression, can be optimized. This is achieved in a similar fashion to the optimization of the other operators. Thus, the rules for empty relations, empty lists of conditions, [true] and [false] can be defined by:

```
srestr (@, [], _, []).                  /*1*/
srestr (@, X, [], X).                   /*2*/
srestr (@, X, [false], []).             /*3*/
srestr (@, X, [true], X).               /*4*/
```

In plain English, rule 1 states that a restriction on an empty relation always evaluates to the empty relation. Rule 2 states that any relation X, except for X equals the empty relation (since rule 1 is defined earlier on), when restricted on an empty list of clauses evaluates to X. Rule 3 states that any relation X restricted on [false] will produce the empty relation. Based upon similar logical reasoning, rule 4 states that any relation X, except for [], when restricted on [true] will deliver X.

Other rules of optimization, including many of the ones proposed by Pecherer and Palermo, as already mentioned, have also been incorporated into ADIM, by use of techniques similar to those discussed above. Thus, for instance, the rule

```
s(@, R1:*:R2, X, Z1:*:Z2):-
        s(@, R1, X, Z1)
        s(@, R2, X, Z2).
```

distributes restriction over join operations, so that relations R1 and R2 could be restricted before performing the join. This, as it is well known, would reduce the size of R1 and R1 prior to the join, so achieving a much more efficient evaluation of the original expression.

Another example of general rules for altering the

order of evaluation within the expression, is the rule

$$s(\hat{}, X, Y@Z, X@Z\hat{}Y).$$

which pushes all restrictions on a relation, to the left, and all projections to the right. In this manner, by subsequent use of the associative rules for restriction and projection, all the restrictions on the relation as well as all the projections could be reduced to one restriction on the relation, followed by one projection.

A most efficient evaluation of this expression can be achieved then, by evaluation of both operation, restriction and projection, on one pass over the given relation.

5.7 Some comments on decomposition -

In this chapter, decomposition techniques have been discussed from a number of different perspectives. Examples of their relevance to the areas of logical design, efficiency, security and integrity of data bases were given. The discussion has in general been centred around practical problems rather than purely theoretical questions. I felt that the theoretical aspects from which ADIM benefits are well covered in the data base

literature. References to them are given in appropriate places in the chapter. Perhaps, the main contribution of this chapter is the presentation of an implementation framework such that extensions and/or modifications to ADIM are very easy to make. This is important in an experimental system of this type.

In designing ADIM, I have come to distinguish two areas of research problems which needed solving. The first has relatively solid foundations. In this area, new fundamental results are unlikely to be produced. This is particularly true, if one is constrained to conventional hardware architectures. Nevertheless, from an engineering point of view, it still represents a challenge in terms of technological trade-offs. These have to be resolved for each particular application. I feel that most of the implementation of the P-unit falls into this category.

The second area is more of an open question. Study and experimentation of problems in this area are more likely to produce significant results of a general nature. I believe that the decomposition techniques of this chapter are more into this category than the former. They together with the ALFRED hierarchy of languages, provide an excellent framework for experimentation in the fields of retrieval languages, query optimization

techniques, security and integrity of data bases. The implementation issues discussed in this chapter, illustrate this point. A specific example is ALFRED-VC's representation of axioms for functional dependencies [section 5.6] and the deduction rules of optimization associated with them [sections 5.4.1, 5.5 and 5.6].

The next chapter, concentrates on the discussion of engineering aspects in the construction of a highly efficient P-unit. This is done within the general principles and aims outlined in Chapter 2.

CHAPTER 6

DATA STRUCTURES AND NATURE OF DATA

6.1 Efficiency -

The processing of a query in ADIM involves the activation of three stages: the decomposition-optimization cycle, the processing of queries involving the basic relations and the composition of a reply. The problems of the second stage and their solutions in ADIM are discussed in this chapter. Maybe the most important problems of this stage are those of efficiency in retrieval operations. My aim was to provide ADIM with a mechanism such that a most efficient retrieval capability could be attained with not too much wastage of secondary memory. This strategy assumes that the cost of secondary memory is of less importance to users than the waiting time for a reply. I believe this to be a reasonable assumption considering the trend to declining prices of memories in the past and the

foreseeable future. This approach to efficiency necessarily leads us into a discussion of the data structures and access mechanisms provided by ADIM.

The access mechanism to the stored data is a set of functions implemented at a low level in the ADIM system. These functions provide a mechanism which is independent of the operating system and/or hardware in use. This approach ensures a high degree of portability for the applications written on top of ADIM as well as for the ADIM system itself. As far as retrieval of data is concerned, the most important problem to solve is the transformation of elements in the data space as seen by the user to the address space provided by secondary memory in the host computer or computers. For the sake of simplicity, I will be referring to the singular computer, where extensions to the ideas exposed are obvious. Otherwise, ideas and principles will be explicitly exposed.

Some authors [HELD75, PECHERER, HELSTO75] have proposed some desirable conditions that the function described above should meet. G. Held and M. Stonebraker in their 1975 paper [HELSTO75], proposed:

"Condition 1.

The function should not introduce additional secondary accesses in order to compute an address."

"Condition 2.

The function should map the given sample of the key space [data space as seen by the user] uniformly across the address space."

The first condition makes the very realistic assumption that in existing commercially available computers, as well as in computers coming on to the market in the foreseeable future, computations performed on data available in main memory are several order of magnitude cheaper in time than the retrieval of data from secondary memory into main memory. In other words, it is more efficient to calculate the address of some data than to look it up in a dictionary held in secondary memory.

Condition 2 establishes the principle that overflow areas should not be used. Obviously, if keys are used to find data in the address space, the extensive use of overflow areas will render key usage as almost useless and unnecessary.

The authors of [HELSTO75] also formulated a third

condition:

"Condition 3.

The function should be an order preserving function (i.e. if K' < K" then H(K') < H(K")) [K' and K" belong to the data space and H(K') and H(K") belong to the address space]".

The purpose of this condition is to provide efficient retrieval in queries of the form

```
RETRIEVE employee WHEN
        [dept = "marketing"]
INTO marketing ?
```

as well as in queries of the form

```
RETRIEVE employee WHEN
        [salary > 10000]
INTO highpaid ?
```

In the first query, a randomizing (hash) function will in most cases provide a very efficient solution. However, the same function will be absolutely hopeless in the second query, since a complete scan of the relation employee will be required. Condition 3 seeks a function able to behave well in both types of queries.

Although the imposition of the above conditions seem to have produced good results in the design and implementation of INGRES [SWKH76], the additional requirements of compactness, modularity and portability impose further conditions to the design of ADIM.

Thus, I have added

Condition 4.

> The selected function should not co-exist with any other function intended for the same purpose.

Condition 5.

> The function should be effective regardless of time dependencies of the data set.

Condition 6.

> The function should have a cost prediction element in it.

The aim of condition 4 is to achieve a high degree of compactness, so making duplication of effort unnecessary. The use throughout the entire system of a unique file structure makes implementation simpler and produces a more compact system than otherwise obtainable with the use of a range of files structures. I feel this makes

the difference on whether the ADIM system will run or not on an 8-bit micro-computer. As an additional bonus, in a small system 'bugs' are better controlled and more easily cured.

Many functions can provide a remarkably good performance in a stable environment where predictions can easily be made, but certainly this is not the case of many applications requiring the use of a data base management system. For instance, booking systems are a case of management of very volatile data where predictions about the shape of the distribution function for this data are difficult to make, if not impossible. Condition 5 seeks a function which is efficient under volatile and stable environments. I believe that performance on retrieval should be independent of the length of time that data resides in a given data base.

Condition 6 assumes that query optimization sub-systems are an integral part of relational data base management systems. A query optimization sub-system based on statistical analysis of past behaviour can only make cost predictions in environments with stable data and queries meeting a certain regularity. I do not believe this to be the general case. Furthermore, the software to collect statistics on the traffic of data and the software to analyse this statistical data can be very

bulky, so making the optimization sub-system too big to fit in small computers where it is most needed. Precisely because of these considerations, I feel that the provision of cost parameters should be a condition of the function under consideration.

Obviously, any function can meet condition 4, if all the other conditions are dropped. Since this is not the case, I had to seek a unique function which will comply with all the other five conditions, as well.

Randomizing (hash) functions [KNUTH] meet conditions 1, 2 and 5, but fail to meet conditions 3 and 6.

Static directories of the ISAM [IBM66] type comply with conditions 2 and 3, but not the others. The variety of functions of this type used by INGRES [HSW75, HELSTO75], called generalized directories, attempts to provide a compromise. They offer tuning parameters for re-organization of directories, whenever efficiency decreases beyond acceptable levels. In this manner, generalised directories fully comply with condition 3 and partially comply with condition 1 and 2. They make no attempt to satisfy conditions 5 and 6.

Unwillingly, to compromise on the failure of the above functions to comply with all six conditions, I

embarked on the study of three more functions. One of these functions attempts to provide a continuum between randomizing functions and directory structures, while the other two concentrates on the problems of volatile data. A discussion of these functions follow.

6.2 Random Directories -

Generalized directories [HELD75] as found in the INGRES access methods represent a continuum of functions between simple order preserving functions at one extreme and normal directories of the ISAM type [IBM66] at the other. It appeared plausible that a continuum between randomizing functions and simple order preserving functions existed. I wanted to investigate this possibility and consequently, I embarked on the study of a class of functions which appeared to be a good candidate. Linear transformations between vector spaces have a sound theoretical basis as well as being general enough to cover a very wide spectrum. Since they had to be implemented with subsequent change in mind, a flexible approach had to be adopted and the particular case of functions based on binary cyclic codes [Appendix G] was chosen.

In order to conduct a minimum set of experiments, I

decided to implement some basic algorithms for coding and decoding binary cyclic codes. This was done by software simulation of different existing hardware devices, an acceptable family of algorithms was recognized and implemented as standard coding and decoding procedures. An example of these algorithms is given in Appendix F.

Cyclic codes are most easily implemented by using shift-register devices. Software for the encoding dictionary is minimized by making use of the cyclic property and the property that each code polynomial is a multiple of the generator polynomial.

Once the basic procedures were established, I started work on the allocation of randomizing functions to regions of the total address space. The regions are not necessarily disjoint. In this manner, the order of the data space is to some extent preserved in the address space. Thus, a reduction of the searching time in queries involving order becomes possible.

Moreover, I also experimented with a dynamic allocation of randomizing functions to regions. This dynamic allocation allows for reorganization of individual regions of the total storage holding a relation.

The use of this technique is complemented by the fixing of a maximum boundary to the access factor and minimum and maximum boundaries to the occupancy factor. Certainly, the access factor and the occupancy factor will not be optimal (except in extraordinary circumstances), but failure of an existing randomizing function to comply with the predetermined boundaries will never be disastrous, because a new member of the family of functions can be selected and the failing region can be reorganized. The techniques for fixing boundaries for the access and occupancy factor are an appropriated modification of the one used by D.G. Held [HELD75] in his generalized directories.

Basically, the scheme as described, is a compromise between randomizing functions and static directory structures. The aim of the scheme was to improve on the performance of a scheme used by INGRES [HELD75]. The results of the experimentation were not particularly encouraging. Although, the reallocation of a function to a region, and the subsequent re-distribution of data onto a larger or smaller region was performed in a reasonable time (in relation to the size of the key space), the amount of information needed in memory, at all times, made this scheme unacceptable. Information needed to keep in memory included, among others: identification of function and its characteristics, parameters, data types

of keys, and boundaries of region.

6.3 Extendible Hashing -

As mentioned earlier on in this chapter, for a number of years in the past, static directories have been used in data base implementations with a relative degree of success [IBM66, HSW75, HELSTO75]. My main objection to their use in ADIM is the constraint imposed by condition 5. The functions implementing an access mechanism for a system based on static directories generally provide very good performance in environments where predictions can be made about the total volume of data, the distribution functions for the keys and the traffic of data as a function of time. Unfortunately, this is not the case in many data base applications, where the nature of the application involves the manipulation of highly volatile data. In this later case, predictions about the shape of the distribution function for the keys, data traffic or volume of data are very difficult if not impossible to make.

However, in recent years, a number of researchers have been exploring schemes for structuring data whose volume is allowed to grow and shrink by large factors [BAYER, COMMER79, FAGIN, TAMMINEN, HOPCROFT83]. The

schemes proposed have gradually converged into two main schemes. The first of these schemes is known as B-trees [BAYER, COMMER79] and we shall be discussing it in Section 6.4. The other scheme, known as extendible hashing [FAGIN], is the main topic of discussion in this section.

Extendible hashing offers a very attractive alternative to the access methods previously discussed in this chapter. It always uses two disc accesses for each search, while at the same time, retaining a capability for efficient insertions and deletions. Remarkably, these characteristics are valid with static and volatile data.

The method was developed in 1978 by R. Fagin, J. Nievergelt, N. Pippenger, and R. Strong [FAGIN]. It is based on an extension of radix search trees, also known as digital search trees or tries [FREDKIN]. Fagin's method attempts to exploit the speed of radix search trees without having to pay the high cost in memory space which characterize the latter.

In general, extendible hashing can be depicted as two files: a directory file and the leaf pages file. The file for the leaf pages store the data. The directory file contains 2^d entries one for each d-bit pattern. A

leaf page contains all data records such that their keys begin with a specific bit pattern. Thus, to search for the record associated with a given key, the leading d-bits of the key are used to index into the directory. This entry of the directory, in turn, stores a pointer to the leaf page associated with the d-bit pattern of the given key. Then, the referenced leaf page is accessed and searched for the proper record. A leaf page can be pointed to by one or more directory entries. If a leaf page holds all the records with keys beginning with a specific k bits, then the directory will have 2^{d-k} directory entries pointing to it.

INSERTION -

To explain the insertion algorithm let us start with a given initial structure. A directory file with only one entry. This entry points to an empty leaf page. A leaf page can hold up to four records. See figure 6.1, below.

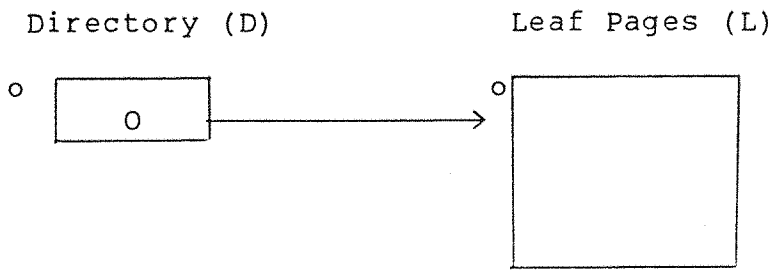


Figure 6.1

Now, let us insert four records. The keys for these records, in binary, are: 01001, 00101, 10100 and 01001. They are placed into leaf page 0:

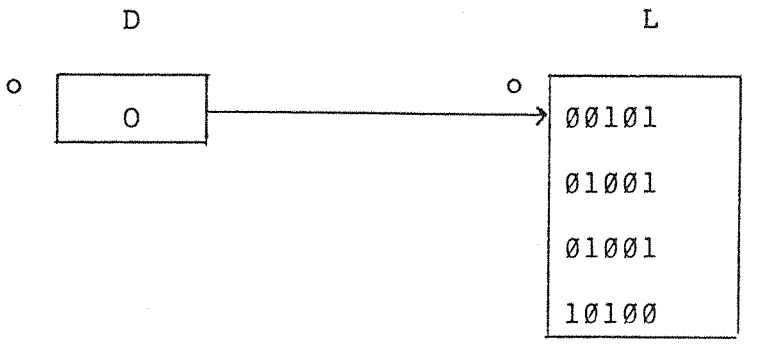


Figure 6.2

The entry in directory D, states that all records are stored in page 0 of the leaf pages file, L, where they are kept in sorted order of their keys. Now, we attempt to insert a new record. Say, the key for this record is in binary notation: 11000. Since, page 0 is full, it must be split to make room for this new record. To do this, we create a second leaf page at the end of file, L. Then, we leave records with key beginning with 0 in page 0, and move those records which key begins with 1 to page 1. The directory size is also doubled in size, thus that a new entry pointing to the new leaf page, can be created. We are left with the following structure:

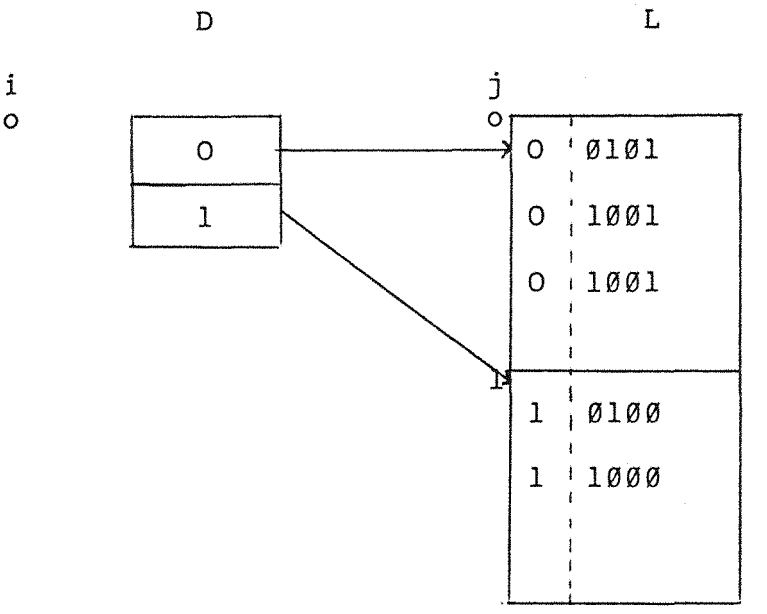


Figure 6.3

We now add the record with key 01010:

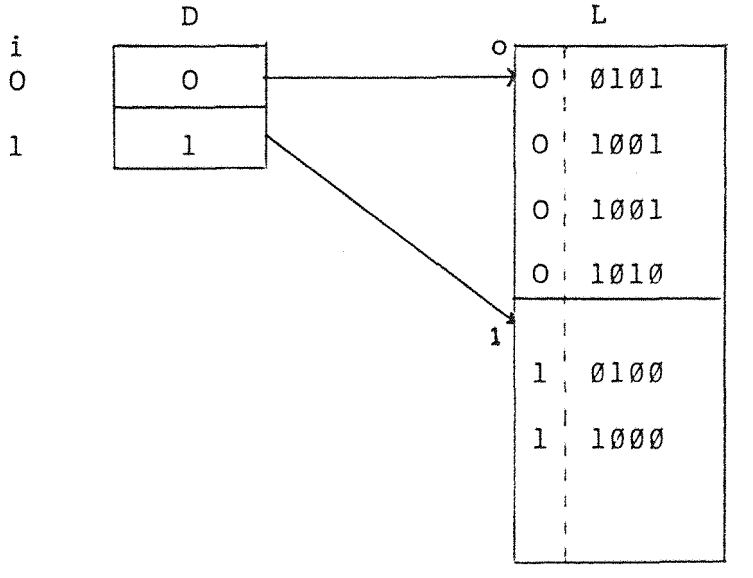


Figure 6.4

In order to add yet one more record, say the record with key 01001, we need to split page 0 again. This time

we will use the two leading bits of the key to index into the directory, D. Thus, we split page 0 into two pages: one page for those records which key starts with ØØ, and one page for those records which key starts with Ø1. It is in the handling of the directory, where Fagin's method differs from more conventional methods. Instead of just creating one more directory entry, pointing to the newly created page (by the split), Fagin's method doubles the size of the directory. Thus our example, becomes:

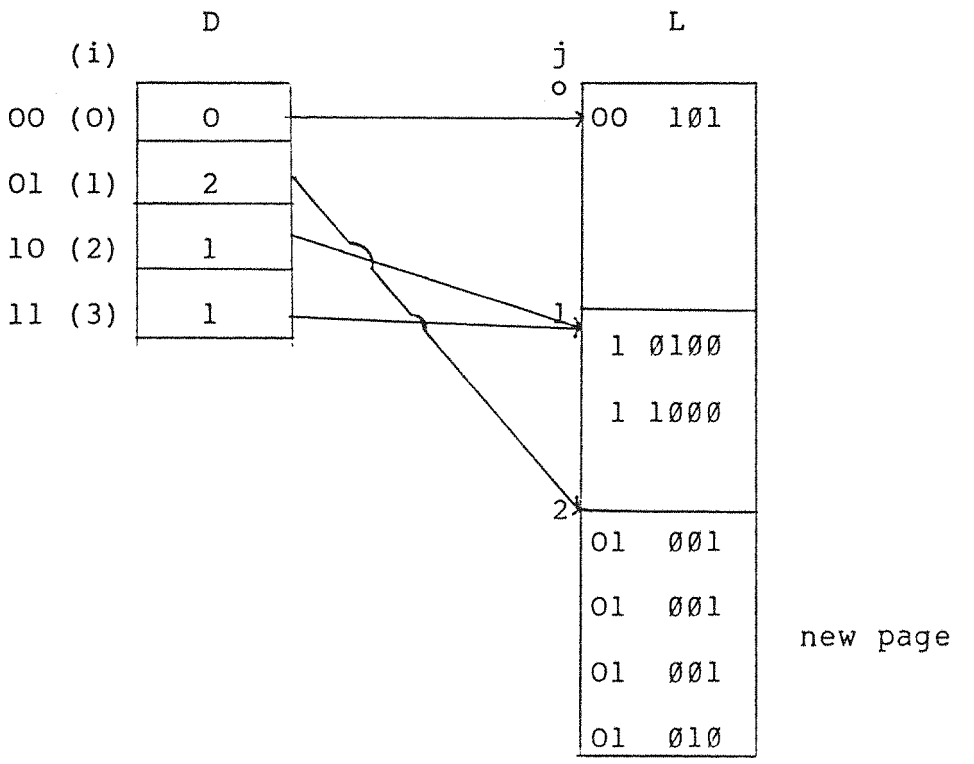


Figure 6.5

SEARCH -

Now, we can access any record by using the leading 2 bits of its key as an index into the directory D. The directory entry i, in turn, holds the number of the page in which the wanted record is stored.

DELETION -

The principles for the deletion algorithm are similar to the insertion algorithm, just discussed. Whenever, empty pages arise after deletions, the algorithm for deletion halves the size of the directory.

As the discussion above clearly demonstrated, extendible hashing provide a graceful and efficient mechanism to handle highly volatile data. But, it should be appreciated that it is not free of problems. The algorithm for insertion, as presented above, is very susceptible to a poor key distribution. The value of d (the leading bits in the key) is the largest number of bits needed to assign keys to leaf pages. Now, if input keys are clustered, large numbers of keys will agree in a large number of leading bits. This will cause a very large directory. In fact, in some applications, the directory could get unacceptably large. A solution to

this problem is to apply a randomizing function (hashing) to the keys, to make them pseudo-random. From this point of view, we can think of the algorithm for splitting nodes as a mechanism to handle hash value collisions. It is this view which earned this method the name: "extendible hashing".

The introduction of hashing, on the one hand, often solves the problem of large directories caused by clustering in the input keys. On the other hand, it re-introduces one of the main problems of randomizing functions, i.e. a very poor performance in range type searches. Some suggestions have been made towards a solution to this problem. Fagin, et.al. [FAGIN] suggested the use of order preserving randomizing functions. As it is well known, these functions in most cases fail to break clusters. To use them for general purposes, the best that we could expect is some reduction in the size of the directory. The magnitude of the reduction would depend on specific applications. Unfortunately, one can expect cases where the reduction in size of the directory will not be significative enough.

A more serious problem with extendible hashing arises when there are more equal keys than the capacity of a leaf page, allows. In this case, the algorithm breaks

down completely. In our example, consider the case where we want to insert two records with key 01001.

In summary, extendible hashing is an excellent scheme for structuring data whose volume is allowed to grow and shrink by large factors. Unfortunately, its suitability to handle data which keys are clustered, present some problems not easily solved in the context of a general purpose system like ADIM. It should also be said, that after I started work in the implementation of an access method for ADIM, some researchers have proposed variations on the extendible hashing scheme that appears to be very hopeful for a general solution of the search by range problem [LITWIN78, LITWIN81, TAMMINEN, LARSON]. Some other interesting ideas about a solution to this problem, can be found in a paper by W.A. Burkhard [BURKHARD]. This paper addresses the more general problem of partially specified queries, and precedes the one by Fagin, et.al.

6.4 Dynamic Trees -

A number of alternatives to extendible hashing have been proposed. In fact, many of these proposals preceded it [BURKHARD, BAYER]. Perhaps, not surprisingly, the most important of these alternatives is one based upon

directory structures which expand and shrink dynamically with usage. I turned my attention to them very early on the life of the ADIM project. After careful consideration of a number of variations based upon 2-3 trees [HOPCROFT83], I decided to provide ADIM with a directory structure based on a generalization of 2-3 trees, known as B-trees [BAYER, COMMER79]. The reason for this decision was the desire to preserve within ADIM's file structure the advantages of static directory structures, e.g. average depth of Ø(log n) for a "random" tree of n nodes, while at the same time, avoid the problems caused by unbalanced trees (directory structures), likely to occur in relations with a high rate of insertions, deletions and updates. Additional reasons for choosing B-trees are provided later on in this chapter.

Before proceeding any further with the discussion about the motivation behind the choice of B-trees, as the unique file structure for relations in ADIM, a clarification of our conceptual framework is required. A closer scrutiny of the terms and concepts embodied in 2-3 trees and B-trees is necessary, if only for the sake of completeness in the exposition. Thus, in the next paragraphs, a brief introduction to 2-3 trees is followed by a more thorough discussion of B-trees. This later discussion starts with a definition of B-trees and an

exploration of their main features. It proceeds to analyse how well B-trees in general, fulfil ADIM's requirements; then, it describes the particular implementation for ADIM. Finally, it re-links with our discussion about the motivation behind the choice of B-trees for ADIM, and draws conclusions about performance, size and complexity of the implementation, and general fulfilment of the conditions outlined earlier on in this chapter.

It should be noticed that we are interested here in the storage of records in files, where the files are stored in blocks of external storage. Hence, the correct interpretation for the idea of a tree, is to think of the nodes as physical blocks. In the sequel, I shall use the word page to refer to a physical block of external storage. Also, since we are dealing with ordered sets, I assume that each record of a file has a key, a set of fields that uniquely identifies each record. For example, the same field of the employees file might be considered such a key.

6.4.1 2-3 Trees -

Static directories based upon a tree structure provide an attractive average depth of Ø(log n).

However, in practical applications, cases often arise of trees with imbalanced growth, and therefore, with branches growing well beyond the average mark. In the case of directory structures, this situation appears as an uncontrolled proliferation of overflow pages. Obviously, this is a situation to be avoided if an efficient retrieval system is to be supported. This suggests that a reorganization of the tree after insertions and deletions might solve the problem. Unfortunately, even in data bases with relations of a moderate size, this is not practical, because of the excessive balancing overhead. An alternative approach to this problem, is to seek a general criterion for controlled growth. One such criterion is embodied in 2-3 trees, and it can be stated as follows:

(a) Interior nodes of the tree can only have two or three children.

(b) All paths from the root to the leaves must have the same length.

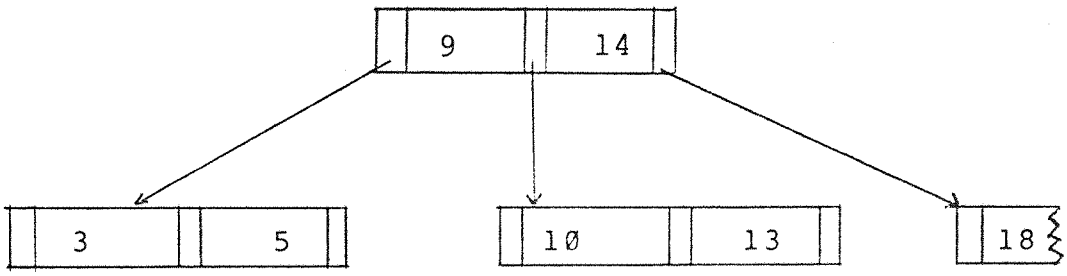


Fig. 6.6

Figure 6.6 is an example of a 2-3 tree. Observe that a 2-3 tree of i levels has between 2^{i-1} and 3^{i-1} leaves. From a different perspective, a 2-3 tree with n elements requires at least $\log_3 n$ levels and no more than $\log_2 n$ levels. Thus, path lengths in the tree are $\emptyset(\log n)$.

The algorithms to insert, delete, update and test for membership of elements in 2-3 trees are suitable adaptations of the corresponding algorithms for binary trees. Since, a binary tree has up to two children per node, the algorithms have to be modified to accommodate up to three children in each node of the tree. Also, deletion and insertion of elements can lead to situations that need special treatment. One such situation arises when an attempt is made to insert a new element in a node with two elements in it. In this case, a split of the node into two nodes is necessary in order to maintain a balanced tree. Another exceptional situation arises when in a node with one element in it, an attempt is made to delete this element. Again, in order to keep the balance of the tree, two adjacent nodes have to be merged together into one node. A generalized version of the algorithms and their handling of special cases is provided by a kind of tree data structure called B-trees [BAYER]. They are discussed in the next sub-section.

6.4.2 B-Trees -

A generalization of the criterion embodied by 2-3 trees was postulated by R. Bayer [BAYER] in 1970:

> "... every page (except one) contains between n and 2n nodes [elements] for a given constant n."

This generalization of the criterion for 2-3 trees is obviously a better criterion for external storage. For a relation with a given number of elements (tuples), an increase in the number of elements per page would normally cause a reduction in the number of pages required to store this relation in external storage. Hence, a tree with fewer levels can be constructed for this relation, so reducing the number of pages to inspect during searches. Put another way, a B-tree is a special kind of balanced tree that permits the retrieval, insertion and deletion of records from an external file with a guaranteed worst-case performance.

Formally, a B-tree is a tree with the following properties:

(a) Each page, except for the root contains at most 2n items.

(b) Each page, except for the root and the leaves, has between n+1 and 2n+1 children.

(c) The root is either a leaf or has at least two children, i.e. one item.

(d) Each path from the root to a leaf has the same length.

Note that a B-tree with n=1 is a 2-3 tree. In general, n is said to be the order of the B-tree.

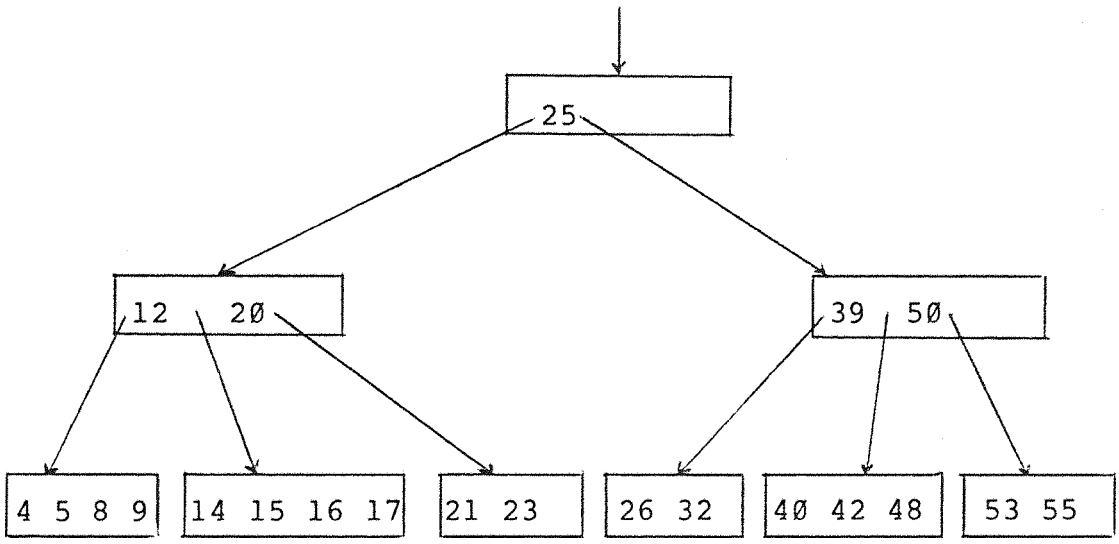


Fig. 6.7 B-tree of order 2.

In a B-tree, we can view a page with m keys, as having the form

$$[p_o, k_1, p_1, k_2, p_2, \ldots, k_m, p_m]$$

where p_i is a pointer to the i^{th} child of the node represented by this page and k_j is a key; $0<i<m$ and $1<j<m$. The keys within the page are in sorted order, so

$k_1<k_2<..<k_m$. In the subtree pointed by p_o, all keys are less than k_1. The opposite is true at the other end of the page, in the subtree pointed by p_m, all keys are greater than k_m. However, in the general case, where $0<i<m$, keys in the subtree pointed by p_i are greater than k_i and less than k_{i+1}.

RETRIEVAL -

To retrieve a record r with key value x, we trace the path from the root page to the page which contains the record r, if it exists in the file. We trace this path by successively fetching pages from external storage into main memory and finding the position of x relative to the keys $k_1,k_2,...,k_m$. If in the latest page brought into main memory, there is a k_i such that $k_i=x$, we have found the record r. Otherwise, if $k_i<x<k_{i+1}$, we next fetch page p_i and repeat the process; if $x<k_1$ we continue the search in page p_o; if $x>k_m$ we use page p_m to continue our search.

INSERTION -

To insert a record r with key x into a B-tree, we first find the page P at which r should belong. If this

page has m<2n records (items), we insert r into this page in the proper sorted order. In the case where m=2n, i.e. page P is full, we would need to change the structure of the tree. To understand what happens in this case, refer to Fig. 6.8. In this example, a record r with key B is inserted in a B-tree of order 2.

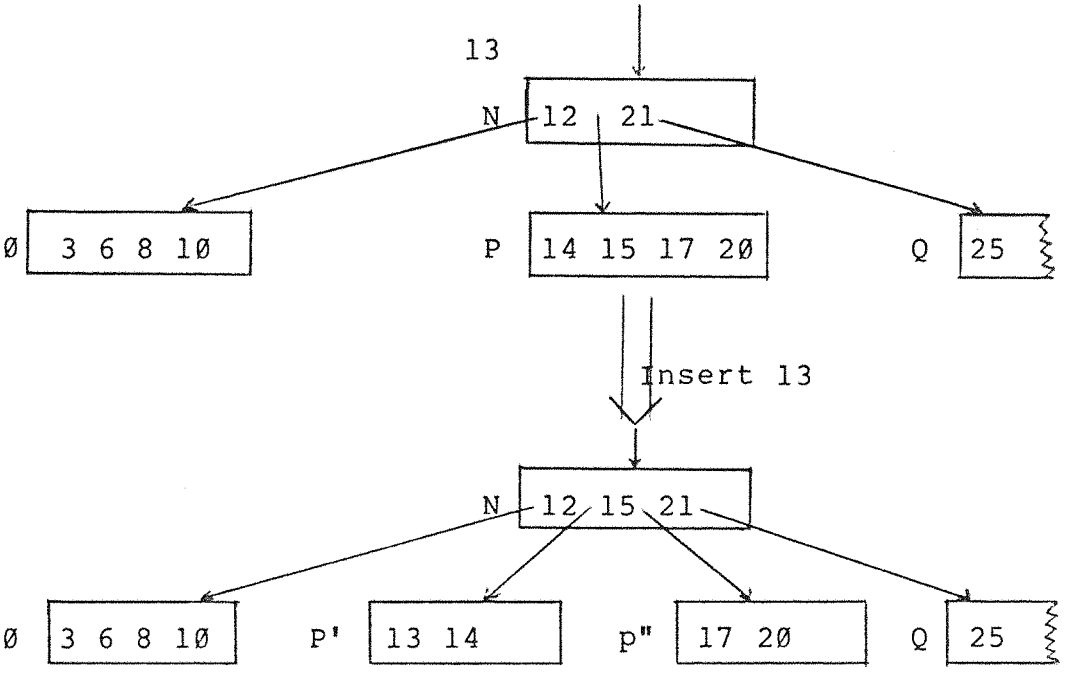


Figure 6.8

The procedure to insert record r with key 13 is:

(a) Key 13 is searched for and not found. The record r should be inserted in page P, but this page is full.

(b) Page P is split into two pages, P' and P".

(c) The 2n+1 records, including record r, are equally distributed into P' and P", and the record with the middle key is moved up one level into the ancestor page N.

Obviously, the insertion of the middle record in the ancestor page could again cause a split of a page, i.e. the ancestor page. In this manner, the splitting of pages could propagate all the way up to the root, thereby increasing the height of the B-tree.

DELETION -

In the algorithm to delete a record r with key x from a B-tree, two cases have to be considered:

(a) The record r with key x is on a leaf page; the trivial case.

(b) The record r with key x is not on a leaf page; in this case, the record r must be replaced by one of the two records whose key values are closest to x; these two records, one on each side of r, happen to be on leaf pages, and therefore, can easily be deleted.

In the latter case, assume $x=k_i$. To find one of the key values closest to x, descend down the pointer p_{i-1} and along the right most pointers to leaf page P. The sought record is the one with key k_m on page P, i.e. the furthest right record on P. To complete the deletion of

record r with key x, replace record r by the record with key k_m on page P, and then reduce the size of P by one.

Any reduction in the size of a page, must be followed by a check of the number m of records left on the page. If m<n, property (b) of B-trees is violated. When this underflow condition is detected immediate corrective action must be undertaken.

An underflow of page P is corrected by borrowing a record from one of the neighbouring pages of P. Because of the cost of having to move another page into main memory, this is a relatively expensive operation. Preventive action should be taken to reduce the frequency of this operation. This can be done by moving more than one record at a time into P, whenever possible. Thus, once a neighbour page is brought into main memory, the records on this page and those in P are distributed evenly on both pages.

Obviously, the removal of the middle record from the ancestor page, could again cause an underflow. This in turn, might need the merging of the ancestor page and one of its neighbour pages. In the extreme case, merging could propagate all the way up to the root. Whenever the size of the root page becomes Ø, i.e. m=Ø, it is itself deleted, thus causing a reduction in the height of the

B-tree.

Figure 6.9-a illustrates deletion, case (a); and Figure 6.9-b illustrates case (b).

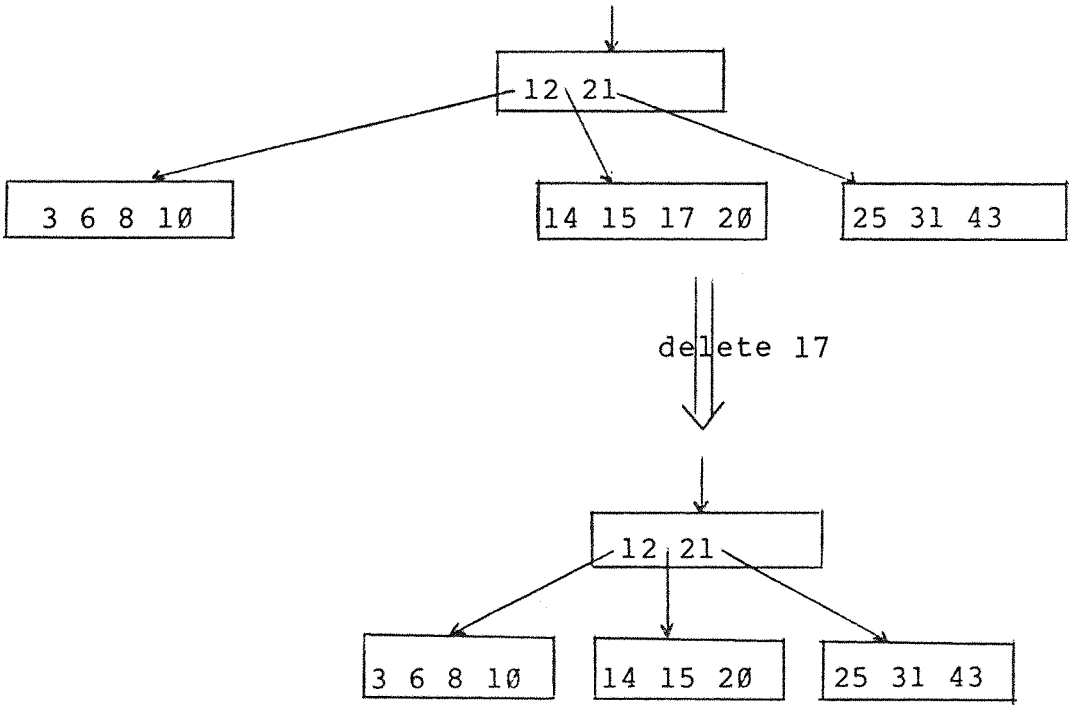


Figure 6.9-a

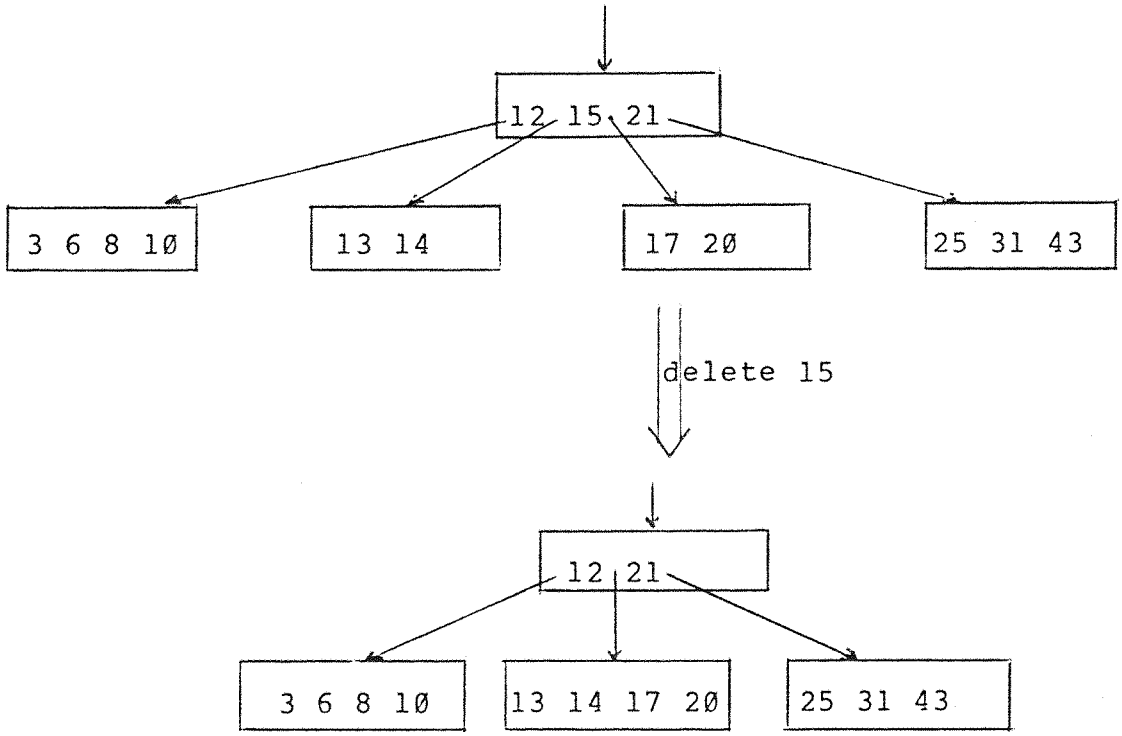


Figure 6.9-b

6.5 B-Tree Implementation -

The implementation of the set of functions that made up ADIM's file management, is now presented. These functions, as well as the rest of the core of ADIM, are written in 'C'. The functions are:

(i) search (): to retrieve a tuple from a B-tree;

(ii) travertree (): to fully traverse a B-tree;

(iii) partial (): to partially traverse a B-tree;

(iv) insert (): to append a new tuple to a B-tree; and

(v) delete (): to delete a tuple from a B-tree.

The definition of the structure of a page precedes the discussion on the actual implementation.

6.5.1 Page Structure -

Typically, B-trees are implemented in two levels: a B-tree for the keys and a flat file for the tuples themselves. The link is established by associating the keys in the B-tree with record positions in the flat file. This type of implementation is illustrated by Figure 6.1Ø.

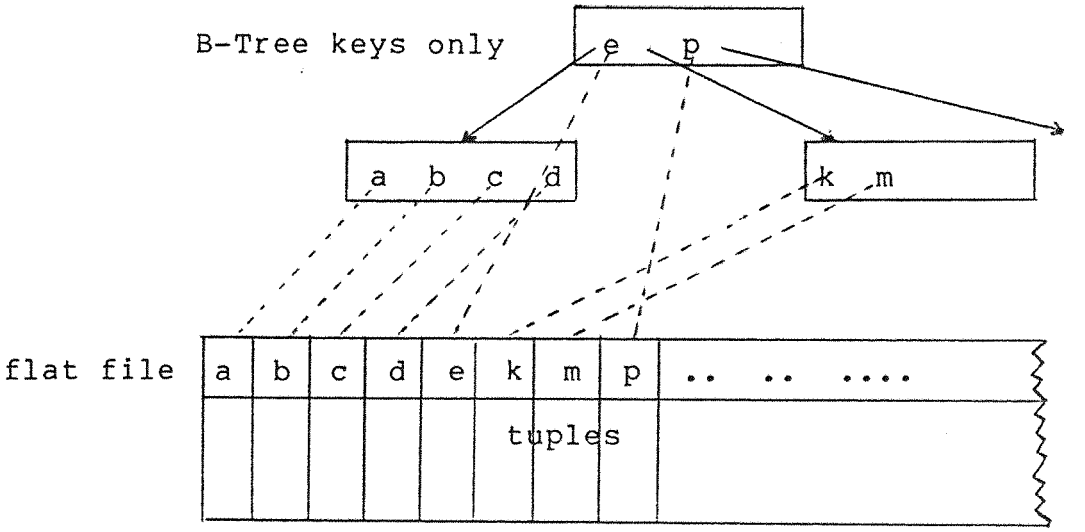


Figure 6.10

The obvious alternative to the above scheme is to store tuples and their keys in the B-tree itself.

For a given page size, a two tier file structure is normally preferred to the one level option. A B-tree restricted to the keys only, would have a higher fan-out ratio than its whole tuples counterpart. This in most cases, would reduce the height of the tree, and consequently, fewer pages would need to be fetched into main memory during searches.

Nevertheless, the one tier option should not be totally discounted without some further consideration. A compromise between the two approaches would be to replicate a two tier B-tree implementation by a B-tree

for whole tuples supported by adequate secondary indexes. Of course, the secondary indexes implemented as B-trees, as well. In this manner, the B-tree for the tuples would appear as the flat file and one of the indexes as the B-tree for the keys. This approach would achieve for the index(es), the high fan-out ratio tree, whenever this is required.

Since, ADIM's expected operational scenario is managing data bases with many relatively small and medium sized relations, I judged the compromise suggested above to have the potentiality for excellent space/time performance, and therefore chose it. It must be emphasised, that this decision was backed up with very conclusive empirical tests [section 6.6]. Large relations are unlikely to be found in a properly constructed ADIM data base, because of the application of decomposition techniques during the process of setting up data bases.

Once the above decision was made, decisions about the structure of a page and ways to represent tuples inside such a page, were painlessly made. Thus, the structure of a page in ADIM was defined by the following sequence of declarations:

```
#define      PGSZ        1Ø24
#define      OFFSET      4*sizeof (int)
#define      PTRSZ       sizeof (int)

struct page
{
    int no;       /*page number*/
    int up;       /*page number of parent*/
    int q;        /*number of tuples in the page*/
    int pØ;       /*extra pointer to child on left*/
    char i_tups[PGSZ - OFFSET];
};
```

Global definitions such as PGSZ, OFFSET and PTRSZ make the porting of ADIM to new machines a relatively easy task. Also, and more importantly, tunning the performance of ADIM is aided by definitions of this sort. For instance, PGSZ which defines the size of a page in bytes, could be set to 512, 1Ø24, 2Ø48 or any other size. Thus, in a computer configuration where the time taken to move 1 byte or 512 bytes from disc to memory is the same, e.g. DEC - PDP11 family, it would be advantageous to define PGSZ as a multiple of 512. More obviously, these definitions also make programs clearer.

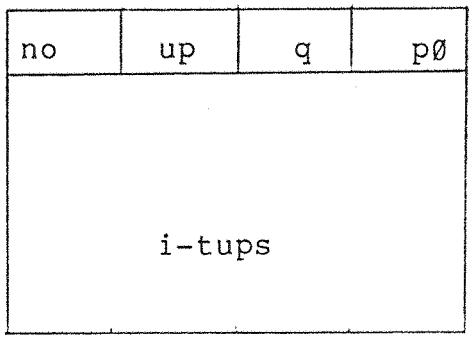


Figure 6.11

The field no is used to stamp the page with a unique identifying number; up refers to the identifying number of the ancestor page (except for the root page); q is the number of tuples stored, at present, in the page; and, if the page is not a leaf, pØ points to the root page of the left most sub-tree. A pointer to a page is recorded by storing the unique identifying number of that page. PTRSZ is the number of bytes required to store such a pointer, and it is determined by the host computer. OFFSET is the total number of bytes required by the fields no, up, q and pØ. The difference PGSZ - OFFSET is the number of bytes available on the page, for the storage of tuples.

Tuples within a page are defined by the 'C' declaration:

```
struct i_tup
{
    int pgno;       /*pointer to sub-tree*/
    char t[MAXTUP];      /*tuple proper*/
};
```

MAXTUP is a global definition which sets the maximum size in bytes for a tuple. MAXTUP is normally defined by the expression (PGSZ - OFFSET)/2 - PTRSZ.

In the structure defined by i_tup, the field pgno points to the root of the sub-tree on the right of the

tuple. The array t is the tuple itself (tuple proper). Although, the declaration of t suggests a fixed size array of MAXTUP bytes; in practice, t occupies a considerably smaller size. In fact, the number of bytes used by the tuple t, is determined by the data types of its attributes. Integer attributes occupy ISZ bytes, reals use RSZ bytes, and strings of characters one byte per character plus one byte for the end of a strings marker. ISZ and RSZ are machine dependent and typical values are eight for RSZ and four for ISZ.

In order to manipulate the apparent overlapping of tuples within a page, another 'C' structure is necessary:

```
union record
{     /*treats tuples in two modes*/
      struct i_tup*cooked; /*formatted tuple*/
      char *raw;           /*unformatted tuple*/
};
```

The union record provides two alternative views for a tuple within a page. As raw, the stored tuple (tuple proper + pointer) can be seen as a sequence of bytes without demarcation between pointer and tuple proper. On the other hand, the field cooked of the union, makes the distinction between the pointer and the tuple proper.

The use of these two views of a stored tuple is illustrated by the following piece of 'C' code:

```
srch_del(d,xx,p,...)  /*searches xx in tree 2nd deletes
                        it*/
      .
      .
      .
      char xx;  /*search key*/
      int p  ;  /*page to search*/
       .
       .
       .
{
       struct page *pp;
       union record kaddr;
                .
                .
                .
       pp = salloc(PGSZ);  /*get memory for page*/
       get_page(d,pp,p);   /*retrieve page p into pp*/
                .
                .
                .
       itpsz = <actual length in bytes for tuple +
                pointer>;
                .
                .
                .
       kaddr.raw = pp -> i_tups; /*get first tuple*/
       kaddr.raw += (k*itpsz);   /*jump to tuple k*/
                .
                .
                .
       if (kcompare (d, xx, (kaddr.cooked)->t)==EQUAL)
                        /*found it*/
       else
                        /*continue search*/
                .
                .
                .
}
```

The function srch_del() searches in the relation described by d, the tuple with key xx and deletes it. The call to salloc() allocates memory space for a page; pp records the location of the memory space allocated. Page p is retrieved by get_page() from disc into the location pointed by pp. Once page p is in main memory, we skip all the fields at the beginning of the page, and position ourselves at the location of the first tuple stored on this page:

```
kaddr.raw = pp -> i_tups;
```

then, we move to tuple k in the page, by:

```
kaddr.raw += (k*itpsz);
```

The variable itpsz holds the actual length, in bytes, occupied by the tuple proper and the pointer associated with it. Thus, in order to compare the key xx with the tuple proper (t), we need to ignore the pointer (pgno). This is achieved by:

```
...kcompare(d,xx,(kaddr.cooked) -> t) ...
```

which, as wanted, skips over the pointer to the sub-tree (pgno), and directly compares the key xx with the tuple proper t.

6.5.2 File Management Functions -

Information about active relations is kept in main memory by descriptors. A descriptor is a brief summary of the structure and general characteristics of a relation. Descriptors are defined by the following 'C' structures:

```
struct relation
{
    char relid [MAXNAME];     /*relation name*/
    long relsave;             /*OS time for save*/
    long reltups;             /*no. of tuples in
                                relation*/
    int relwid;               /*width in bytes of rel.*/
    unsigned relattss;        /*no. of atts.*/
    unsigned dvc;             /*device for rel.*/
    int root;                 /*page no of root*/
    unsigned n;               /*n for B-tree*/
};

struct descriptor
{
    struct relation reldum; /*dump of relation tuple*/
    char status;            /*open, closed, etc*/
    unsigned devdesc;       /*ADIM DEV.descriptor*/
    char offset [MAXDOM];   /*offset to att.i*/
    char fmt [MAXDOM];      /*format of att.i*/
    char fl[MAXDOM];        /*length in bytes of att i*/
    char given [MAXDOM];    /*value supplied in key
                              YES/NO*/
};
```

In the structure relation, relid stores the name of the relation; relsave keeps information about the validity date for this relation; reltups keeps track of the number of tuples; relwid is the width in bytes of a tuple; relatts is the number of attributes in the

relation; dvc is the device (disc) on which the relation is stored; root is the page number of the root page in the B-tree; and, n is the degree of the B-tree.

In descriptor, reldum is a replica of the relation above; status knows about the actual condition of the relation: open, closed, etc; devdesc is a machine independent device descriptor; offset[i] is the offset in bytes from attribute[Ø] to attribute[i]; fmt[i] is the data type of attribute[i]: 'c', 'i', 'r', etc; fl[i] is the length in bytes of attribute[i]; and, given[i] records whether a key for attribute[i] has or has not been supplied.

Further information about the attributes of relations are kept in a catalogue which is defined by the 'C' structure:

```
struct attribute
{
    char aname[MAXNAME];     /*name of attribute*/
    char rid[MAXNAME];       /*name of relation*/
    char format[FSZ];        /*integer, real, string of
                               chars, etc*/
    int asize;               /*length in bytes of this
                               att.*/
    int start;               /*starting position in tuple
                               (byte)*/
    int relative;            /*relative position:first,
                               second,etc*/
    int key pos;             /*relative position in key*/
};
```

In attribute, aname is the name of the attribute; rid refers to the relation to which the attribute is part of; format is the data type of the attribute, e.g. c20 - a character string of length twenty; asize is the length in bytes of this attribute; start is the offset, in bytes, from the left edge of the tuple to this attribute; relative is the relative position of this attribute, within the list of attributes belonging to this relation (rid)- i.e. a value between zero and the degree of the relation minus one; and keypos is the relative position in the key - i.e. a value between one and the number of attributes in the key.

Now that the underlying structures of ADIM's file management have been presented, let us examine the set of functions that make up ADIM's File Management Functions. First of all, to operate on a relation, we will need to activate it, and later on, once we have finished with it, we will have to deactivate it.

The function openr() makes a named relation active. Briefly, openr() sets up a descriptor for the relation. This descriptor is set up from information held in the system's catalogues 'relation' and 'attribute'.

As the counterpart to openr(), the function closer() deactivates a relation. For this, it uses the

information in the descriptor to update the catalogues 'relation' and 'attribute'.

Since, the algorithms for openr() and closer() are not of primary importance to the current discussion, I shall not dwelve into them, in this chapter. Further details are given in Chapter 7.

The algorithms for search(), travertree(), partial(), insert() and delete() were, not surprisingly, written in 'C'. Their implementation is a recursive version of the general algorithms described in Section 6.4.2. In this manner, the implementation of search() was based on retrieval, insert() on insertion and delete() on deletion. Obviously, travertree() and partial() are extensions of the algorithm for search(). Thus, travertree() was implemented as a recursive traversal of the B-tree, and partial() was implemented as the functional composition of search() and travertree().

Maybe, some specific aspects of the implementation of these functions needs some further discussion. For instance, setting up frames for the pages and tuples in a particular relation, is a problem that needs to be solved by all of the File Manipulation Functions, with the exception of openr() and closer(). To discuss it, let us consider the function search():

```
int n, nn, tpsz, itpsz;       /*global to this file*/

search(d,t,action,extra,rw)   /*searches t in relation d*/
      struct descriptor *d;
      char *t;                /*pattern to match*/
      int (*action)();
      char *extra;
      char rw;               /*read/write permission*/
{
      /*frame tuple*/
      n = d -> reldum.n;
      nn = 2*n;
      tpsz = d -> reldum.relwid;
      itpsz = tpsz + sizeof (int);

      return(srh_get(d,
                     t,
                     d -> reldum.root,
                     (*action),
                     extra,
                     rw
                    )
            );
}
```

Searching for a tuple to match the pattern in t, is done by srh_get(). But, before srh_get() is called, search() sets up a frame for the tuples in this relation. The degree -n of the B-tree and the length, in bytes, of the tuples -tpsz, are obtained from the descriptor d. The length, in bytes, of the tuple plus the pointer to the sub-tree, are then calculated -itpsz. Similarly, the maximum number of tuples in a page -nn, is obtained by doubling the value of n.

The descriptor d was set up by a previous call to openr(). Likewise, the function setkey(), discussed in Chapter 7, sets the pattern to be sought in t. The

character in rw specify access authorization for this search.

Perhaps the most interesting aspect of ADIM's implementation of B-trees, is the extensive use of functional composition. For instance, in the case of search(), it is difficult to imagine anybody searching for a particular tuple, without a purpose in mind. Normally once a tuple is found, some further processing would take place, e.g. print the tuple. Hence, functional composition, as an integral part of the File Manipulation Functions, becomes a very powerful technique.

In search(), travertree() and partial(), the parameter action is a pointer to a function to be composed with the calling function. The parameter extra, in turn, provides a pointer to the parameters to be used by action. An illustrative example of the use of this technique, is the function printr().

```
/* PRINTR - prints a relation on user's VDU
            A simplified version - no error handling
*/

struct endofmarks
{

    char eof;     /*end of field marker*/
    char eot;     /*end of tuple marker*/
};

/* printr....*/
printr( rel )
    char *rel;     /*name of relation*/
{
    struct markers = {'1', '\n'};
    struct descriptor desc;
    struct descriptor *d = desc;

    if ( openr(d, rel, R) == FAIL) return (FAIL);

    printf ("RELATION:%s\n",rel);

    travertree(d,d->reldum.root, print_tuple, & markers);

    closer (D);

    printf ("\n\n");
}



print_tuple(d,tuple,marks)  /*prints a tuple on user's
                              VDU*/
        struct descriptor *d;
        char *tuple;       /*tuple to print*/
        struct endofmarks * marks;
{
        int i;

        for (i=0; i<d->reldum.relatts; i++)  /*for each
                                               attribute*/
        {
            < print the attribute >;
            putchar (marks->eof);  /*end of field*/
        }

        putchar (marks->eot);     /*end of tuple*/
}
```

First of all, printr() initializes the variable markers, to '|' for end of field and '\n' for end of tuple. If the relation rel is successfully opened by openr(), its name is displayed on the user's terminal. Then, travertree() is composed with the function print_tuple(), to print every tuple of rel. The function print_tuple() uses the markers to print tuple, each time it is invoked by travertree(). Finally, the relation rel is closed by closer().

Thus, assuming the existence of the relation employee(name, salary), the call printr("employee") would produce

```
RELATION:  employee

K.Robertson       |    387.25
T.Hamilton        |    531.15
                .
                .
                .
R.Johnson         |    423.10
```

In the printr() example above, the whole of rel was printed, since travertree() was called with d->reldum.root, the root page for rel. If only a sub-tree of the whole tree storing the relation was to be printed, the root page for that sub-tree should be provided. Thus, in the implementation of partial(), the call to travertree() is preceded by a call to search(),

which finds the root page for the sub-tree.

It should also be noticed that, by using a different "action" function in our printr() example, we could format the display of rel differently. Thus, by changing print_tuple() to an appropriate function, we could use printr() as a general display facility for relations, a report generator, a facility in an integrated DBMS and text processing package, etc.

In a similar fashion, we could use functional composition in arithmetic applications. For instance, to calculate the salary bill of a company, we could use travertree() in the following manner:

```
    .
    .
    .
    total_salary = Ø.Ø;

    travertree(d, d -> reldum.root, add_salary, &
               total_salary);

    printf("TOTAL SALARIES:%f\n", total_salary);
    .
    .
    .
add_salary(d, tuple, tsal) /*add salary to tsal (total
           salary)*/
    struct descriptor *d;
    char *tuple;    /*tuple with salary attribute*/
    double *tsal;   /*total salaries*/
{
    double *psalary;

    psalary = < position in tuple of attribute salary >;
    *tsal += *psalary;    /*add salary to total*/
}
```

In order to examine the implementation of functional composition, let us have a closer look at travertree():

```
/* TRAVERTREE - B-tree traversal
   A simplified version - no error handling.
*/

travertree (d, node, action, extra)
    struct descriptor *d;
    int node;       /*root page*/
    int (*action)();       /*fog*/
    char *extra;
{
    union record tt;
    struct page *pp, *salloc();

    if (node != END)    /*not the bottom of the B-tree*/
    {
        pp = salloc(PGSZ);  /*allocate memory for page*/
        get_page (d,pp,node);    /*retrieve node into
                                   pp*/

        tt -> raw = p -> i_tups;  /*get to first tuple*/

        travertree (d,pp->pØ,action,extra);  /*down pØ*/

        for (i = pp->q; i>Ø; i--)
        {
            (*action)(d,(tt.cooked)->t,extra);
            /*compose*/ travertree (d,(tt.cooked) ->
            pgno,action,extra); tt -> raw +=
            (d->reldum.relwid + sizeof(int));
        }
        sfree(pp);  /*release memory*/
}
```

Travertree() parameters are by now fairly familiar to us, and hopefully, do not require further explaining. Nevertheless, if it is still felt that an explanation is necessary, please, see the search() example earlier on in this section. As for the variables declared internally to travertree(), the union tt of type record, allows us to look at tuples in raw and cooked form, according to requirements. The function salloc() returns a pointer to an area of main memory, capable of storing a page. The variable pp is a pointer to such area of memory.

The first test performed by travertree(), is to check that it has not hit the bottom of the tree. If that was the case, travertree() returns immediately. Otherwise, salloc() allocates main memory for a page node, which in turn, is retrieved from secondary storage by get_page(). Then, the pointer tt is set to point to the first tuple in the page, ready to start processing.

The 'for' loop, iteratively, applies the function action to each tuple (t) in the page. It also calls travertree() recursively, thus the sub-tree beneath each pointer (pgno), could also be processed by action.

```
for (i = pp -> q; i>Ø; i--)
{
    (*action)(d,(tt.cooked) -> t,extra); /*compose*/
    travertree(d,(tt.cooked) -> pgno,action,extra);
    .
    .
    .
}
```

The statement

```
tt -> raw += (d -> reldum.relwid + sizeof(int));
```

reposition the pointer tt to point to the next tuple on the page.

Since, in a page there is one more pointer than tuples, i.e. the pointer pØ, some special action is required, if the sub-tree beneath pØ is not to be ignored. Thus, before entering the loop, an additional recursive call to travertree() is made:

```
travertree(d,pp -> pØ,action,extra);  /*down pØ*/
```

Finally, once we come out of the loop, the memory space occupied by page node, is no longer required, and therefore, it is released for re-use by ADIM. This is done with a call to sfree().

In this example as well as in numerous previous

examples, the function get_page() has been used. This function provides a machine independent interface, between the File Manipulation Functions and the host operating system's file management. In fact, ADIM supports its own device handlers. This, I believe, enhances the portability and efficiency of ADIM. For a discussion of these aspects of the implementation of ADIM, see Chapter 7.

6.5.3 Memory Management -

Perhaps, more closely related to the File Manipulation Functions, are the memory management functions, salloc() and sfree(). These two functions implement a memory management system based on a stack discipline. This technique provides a natural 'cache' memory for the File Manipulation Functions. Empirical support for this assertion is provided by practically all the examples in this section. All of the File Manipulation Functions, except for openr() and closer(), have been implemented recursively, and hence a stack memory is not only sufficient, but also extremely efficient.

To illustrate the argument above, consider travertree() once more. Each call to travertree() gets

memory from the stack, by calling salloc(). Thus, recursive calls to travertree(), gradually increase the height of the stack. Now, just before returning, travertree() releases memory back to the stack by calling sfree(). Consequently, the stack gradually and gracefully shrinks.

By choosing a reasonable page size, and allocating memory space for the stack, commensurate to the page size, a simple and powerful memory management is achieved for each particular application of ADIM.

6.5.4 FML: Comments on implementation -

An appropriate characterization of the File Manipulation Functions is perhaps compactness. One of ADIM's stated objectives is to provide a data base management system for small computers. The code for the File Management Functions, despite its complexity, is extremely compact. It is my belief, that this was only possible because of the extensive use of functional composition and a matching memory management sub-system. This compactness was not achieved at the expense of efficiency. On the contrary, functional composition and the stack memory management positively contributed to the implementation of a highly efficient system.

6.6 Empirical Tests -

Concurrently to the design and development of ADIM, some empirical tests were conducted. These tests were performed, at different stages, during the development of ADIM. The implementation and subsequent operation of two application systems were used as a material base for experimentation. Below, a report on the performance, implementation and use of B-trees by these applications, follows. Details of 'worst' case performance, in both systems, are also given.

The first of the applications, named Commodities Buyer Agency, despite its complexity, still is a good example of an information system with an underlying data base of relatively small size. Relations sizes range between a dozen tuples and up to ten thousand tuples. The second application, an Examinations Monitoring system is interesting because of its larger relations. During joins, Intermediate relations could easily have well over a quarter of a million tuples. The relevance to ADIM of these systems is more than apparent. The implementation of B-trees as used by the Commodities Buyer Agency, the Examinations Monitoring system and finally, ADIM itself, should be seen as progressive refinements of the same basic ideas. It should also be emphasized that both systems used for experimentation, Commodities Buyer

Agency and Examinations Monitoring are today successfully operating on a daily basis.

6.6.1 Commodities Buyer Agency -

This is a system for a company acting as a buying agent for third party companies. For the purpose of this report, I shall refer to the company acting as a buying agent as the agent, and its customers companies as buyers. Companies selling through the agent, shall be referred to as suppliers.

In this system buyers ask the agent for details of prices, delivery date, discounts, etc. available for a given product. This process is called the enquiry. The agent, in turn, asks for quotations from suppliers. The suppliers quotes depend on the number of units being bought, payment terms, delivery time required, possible penalties for delays, commission to be paid to the agent, etc. Only, when all parties - buyer, supplier and agent - reach an agreement, contracts are signed, and then the commercial transaction proceeds. Bad buyers and suppliers are restricted from entering the system. Also, according to their past record, buyers and suppliers are ranked. Thus, suppliers who pay high commissions, deliver and pay commissions on time, are likely to

receive more and better requests for quotations.

Details about enquiries, quotations and contracts are input, deleted and modified, interactively. Statistical reports, bills and contracts are prepared on batch mode.

Shortly after the first implementation of ADIM's B-tree file management was completed, and around the time that the specifications for the commodities system were being prepared, the data base management system DBaseII [ASHTON] was released. DBaseII claimed (and still does) to be a relational data base management system for small computers [8 bit microcomputers]. Moreover, DBaseII was the first commercial system of its kind to offer B-trees in its file management subsystem.

In many respects, one could find similarities between DBaseII and the kernel of ADIM, at least, on paper. DBaseII and ADIM are relational systems for small computers, and both use B-trees in their file management. Because of these similarities, an early evaluation of DBaseII was highly desirable. Thus, we chose it for the implementation of the Commodities Buyer Agency system.

In general, a relatively quick implementation was possible. The whole system was implemented within three months. The hardware used for the system was:

COMMART Communicator: 8-bits micro running the MP/M operating system;
64kbytes of memory + 4*48kbytes of memory;
18 Mbytes Winchester disc;
2 8" floppy discs;
4 terminals; and
1 printer.

The relations for the most important entities in the system are:

enquiries - holds details about enquiries;
quotations - quotes received from suppliers;
contracts - main details of contracts; and
payments - to monitor outstanding payments.

The implementation and subsequent operation of the Commodities Buyer Agency system, produced the following finds:

Positive

1. A relatively quick implementation. The whole implementation of the Commodities Buyer Agency system took six month/man.

2. An implementation easy to understand by non-computer specialists. Nowadays, the commodities system is run and maintained by personnel, who at the time of the implementation, had no previous computer knowledge.

Negative

1. DBaseII's explicit two levels implementation of B-trees, i.e. one sequential file for the relation and B-tree files for the indexes, confuses inexperienced users. For instance, expressions such as:

   ```
   USE enquiries
   USE enquiries INDEX enqndx1
   USE enquiries INDEX enqndx2
   ```

   are not clear to users, unless they know about indexes and also, understand the way that 'enqndx1' and 'enqndx2' were built.

2. Moreover, expressions as the ones above, are absolutely contrary to what is regarded as one of the characteristics of relational systems, i.e. the separation of the logical view of data from the physical details of the implementation. Another example of this, is the explicit use of memory partitions, e.g.

   ```
   SELECT primary
   SELECT secondary
   ```

   Even worse, the user must know whether indexes are being used or not. The command FIND, which searches an indexed relation, would produce unpredictable results if used on a relation with the wrong index or no index. For a relation without indexes LOCATE could do the same as FIND (!). In addition, the syntax for the 'boolean' condition in FIND and LOCATE is different.

3. Nevertheless, if JOIN did work on a multi-user environment, there would have been less of a need to use the non-relational operators FIND and LOCATE. Unfortunately, JOIN and SORT do not always work with relations bigger than 1ØØ tuples, under MP/M.

4. More seriously, DBaseII's implementation of relations on sequential files and indexes on B-tree files, means that with volatile data, extensive re-organization of multiple indexes often has to be done. Deleted tuples are not erased from the sequential file, neither from the B-tree. In order to really delete them, PURGE has to be used and the B-tree for the index has to be re-constructed. It makes one wonder why B-trees were used in the first place, when a static directory structure would have done exactly the same job without the added complexity of on the 'fly' re-structuring of data files. Does DBaseII really support B-trees? Let us believe that it does.

In summary, the relational capabilities of DBaseII was found to be rather restricted. Nevertheless, the use of DBaseII in the commodities system still allowed me to test the performance of B-trees. For this purpose, possible extreme cases were considered and two such situations identified. The first case, was the interactive retrieval of one tuple, and the second case, a join involving two large relations. Obviously, relations sizes are relative to the size of the computer in use.

Although an indexed search for a particular tuple may take some time, the performance is still satisfactory.

At the other extreme, performance was assisted by simulation of a JOIN. This was necessary, since JOIN does not work properly under MP/M. The structure of the relations used and the program for this simulation

follows:

```
STRUCTURE FOR FILE:   B:ENQUC0   .DBF
NUMBER OF RECORDS:    00600
DATE OF LAST UPDATE: 00/00/00
PRIMARY USE DATABASE
FLD        NAME        TYPE  WIDTH    DEC
001      ENQ:DATE       C     008
002      ETCOENQ:N      C     007
003      CLNENQ:N       C     020
004      SUP:NO         C     006
005      CLN:NAME       C     015
006      DATE:TOSUP     C     008
007      TRD:NAME       C     015
008      STATUS         C     020
009      SUP:NAME       C     030
010      COUNTRY        C     020
011      QUOT:DATE      C     008
012      DATE:TOCLN     C     008
013      CLN:ANS        C     008
014      CONT:DATE      C     008
015      CLNCONT:N      C     025
016      REMARKS        C     020
** TOTAL **                 00227
.
```

```
STRUCTURE FOR FILE:   B:QUOT     .DBF
NUMBER OF RECORDS:    10146
DATE OF LAST UPDATE: 00/00/00
PRIMARY USE DATABASE
FLD        NAME        TYPE  WIDTH    DEC
001      ETCOENQ:N      C     007
002      SUP:NO         C     006
003      QUOT:DATE      C     008
004      MONT:UNIT      C     003
005      QUOT:VALUE     N     012     002
006      EXP:DATE       C     008
007      DATE:TOCLN     C     008
008      MEDIUM         C     001
009      FORM:PAY       C     007
010      STATUS         C     020
** TOTAL **                 00081
.
```

```
*************************************************************************
*         BUSCAR.CMD
*************************************************************************
CLEAR
SET TALK ON
SET FORMAT TO SCREEN
ERASE
SELECT SECONDARY
USE quot INDEX xquoetco
SELECT PRIMARY
USE enquco
DO WHILE .NOT. EOF
        STORE etcoenq:n TO numero
        SELECT SECONDARY
        FIND &numero
        IF f=0
                SELECT PRIMARY
                SKIP
        ELSE
                STORE quot:date TO uno
                STORE date:tocln TO dos
                SELECT PRIMARY
                REPLACE quot:date WITH uno,date:tocln WITH dos
                SKIP
        ENDIF
ENDDO
ERASE
QUIT
```

Relation 'enquco' had 600 tuples and relation 'quot' had 10146 tuples. The program produces a simple report on the date of quotation and validity of the quote. The test was run with three other users in the system. None of the other users were using DBaseII. The work load of the system, at the time, could be described as light. The total time taken by this simulated JOIN was 31 minutes.

6.6.2 Examinations Monitoring System -

The purpose of this system is to collect the names and other relevant information about students taking a series of examinations. In the current year, each student may register for examinations in a number of subjects, varying between one and fifteen. The registration of the students, some 60-100 thousand per year, is done at their own schools. For administrative purposes, the schools are grouped into local education authorities. Due to the large quantities of data, the registration of candidates, input of examination results, issue of certificates and the production of multiple reports for operational and statistical purposes, are all done in batch mode. As one would also expect on a system of this nature, queries about individual candidates and amendments to the data relating to them, are normally

done interactively.

A file or data base management system for the Examination Monitoring System, would need to handle data which is grouped in large logical collections and also represents complex relationships.

Preliminary studies on the possible software tools for the implementation of this system, established that no suitable commercially available data base management system existed. One of the specifications for this system, was the use of a Hewlett-Packard minicomputer, which certainly restricted the choice of software tools.

Perhaps, the attraction of a low cost microcomputer based implementation would have persuaded us and the commissioners of the system, to use DBaseII on different hardware. Fortunately, our previous experience with DBaseII clearly demonstrated its unsuitability for a project of such scope and complexity as this one.

Thus, a decision was made to write our own file management module for the Examination Monitoring System. Clearly, this was an excellent opportunity to perform further tests on B-trees, and in particular, ADIM's own implementation of them.

Ideally, I would have liked to put ADIM directly to the test. Unfortunately, a compiler for the 'C' language was not available for this particular machine. Hence, an alternative had to be found. Algorithms identical to the ones used in ADIM, were coded in PASCAL, and subsequently used as the file management module of the Examination Monitoring System.

From the point of view of the ease of use of ADIM's file management, the particular implementation of the Examination Monitoring System, should be of little relevance, since ALFRED was not used. Nevertheless, there is a point which is worth while mentioning. Considering the magnitude of the project, a relatively short period of time was taken for the implementation of the whole system. Including the PASCAL re-write of ADIM's file management, the Examination Monitoring System was implemented in months rather than years. I believe this was possible, mainly, because of the functional composition capabilities of ADIM's file management.

More importantly, once the system was fully tested and had completed its first year of operation, its performance could be evaluated. I believe it could be described as more than satisfactory. For instance, multiuser interactive queries and updates, take a time that for all practical purposes, is negligible. At the

other extreme of the scale, reports generated in batch mode, at worst, only take a few minutes.

To illustrate these worst cases, I have included below two programs from the Examination Monitoring System. The first one, generates a general statistical report on the results obtained by students in their examinations. The second program, groups the candidates by school and then prints their results.

The program 'statsl' completes a full traversal of a large B-tree. This program was run on a machine with no other users on it. The candidates file (CANDREL) had 67,ØØØ valid entries and its size was 12 Mbytes. The program was executed in 434 seconds of CPU time and run during 11 minutes of real time.

```
PROGRAM Stats1( input, output );

{*

DESCRIPTION
    First statistics program, which calculates
      awarded a certificate;
      with grades U and A excluding A only;
      with grades A only;
      withdrawn;

    A list of all candidates for whom one or a
    missing is also printed.

*}

$include 'dbdefs.s'$
$include 'gpglobs.s'$
$include 'stat1defs.s'$


CONST
    SCRATCH         = 'scratch.a';

VAR
    Mawarded,
    Fawarded,
    MUexceptA,
    FUexceptA,
    MallA,
    FallA,
    Mwithdrawn,
    Fwithdrawn,
    total,
    k                       : positive;
    td,
    rd                      : text;

    schoolno,
    currschoolno            : fivedigits;
    subjectno,
    candno                  : threedigits;


BEGIN  {* Stats1 - MAIN *}

    {* initialization *}

    OpenDB( SREB );

    rewrite( td, SCRATCH );  {* work file *}

    Mawarded := 0;   {* init counters *}
    Fawarded := 0;
    MUexceptA := 0;
    FUexceptA := 0;
    MallA := 0;
    FallA := 0;
    Mwithdrawn := 0;
    Fwithdrawn := 0;

    (* Count all different ocurrences of ... *)
    currschoolno := '00000';
    total := 0;
    OpAll( CANDREL, CANDINDX, R, CountResults );


    (* prepare report *)

    rewrite( rd, REPORT1 );

    writeln( rd );  writeln( rd );
    writeln( rd,
             '                S.R.E.B.  EXAMINATIONS CERTIFICATE STATISTICS' );
    writeln( rd,
             '                ********************************************' );
    writeln( rd );  writeln( rd );
    writeln( rd,
                                                          'FEMALE    MALE' );
    writeln( rd );  writeln( rd );

    writeln( rd,
    'NUMBER OF CANDIDATES :-' );
    writeln( rd );
    writeln( rd,
    '     AWARDED A CERTIFICATE                ', Fawarded, Mawarded );
    writeln( rd,
    '     GRADES U & A excluding A only', FUexceptA, MUexceptA );
    writeln( rd,
    '     GRADE A only                         ', FallA, MallA );
    writeln( rd,
    '     WITHDRAWN                            ', Fwithdrawn, Mwithdrawn );

    writeln( rd );  writeln( rd );

    writeln( rd,
    'CANDIDATE/SUBJECTS WITH GRADES MISSING :-' );
    writeln( rd );

    reset( td, SCRATCH );

    k := 0;
    currschoolno := '00000';
    WHILE NOT eof( td ) DO
    BEGIN
        readln(td, schoolno, candno, subjectno);

        IF schoolno <> currschoolno THEN BEGIN
            IF (k MOD 10) <> 0 THEN writeln(rd);
            writeln(rd);
            writeln(rd, 'School: ', schoolno);
            currschoolno := schoolno;
            k := 0;
        END;
        write(rd, candno, '/', subjectno, '  ');
        k := k + 1;
        IF (k MOD 10) = 0 THEN  writeln(rd);
    END;


    {* wind up *}

    close( td );
    close( rd, 'SAVE' );

    writeln( 'Number of Candidates processed ...  ', total )
END.
```

The program 'NTRYSCHS' is interesting, because it implements a join of four relations. The relations involved are:

```
CANDREL - candidates; entries:67,500; size:12.2Mbytes
SCHREL  - schools;    entries:600   ; size:14Kbytes
SBJREL  - subjects;   entries:579   ; size:39.4Kbytes
LEAREL  - leas;       entries:40    ; size:1.6Kbytes
```

The following program was run on a machine with no other users on it. Execution time was 2 hrs. and 15 minutes.

```
PROGRAM NTRYSCHS(input,output);

{*}
{DESCRIPTION:
   Program to print Candidate results by school}

{*}
CONST
   boardname = 'SOUTHERN REGIONAL EXAMINATION BOARD';
   heading = 'STATEMENT OF ENTRIES 19';
   Pagesize = 58;

VAR
   infile      : FILE OF exeayear;
   outfile     : TEXT;
   thisyear    : exeayear;
   reply       : char;
   StartKey    : keyval;

   leacode     : twodigits;
   prevschool,
   schoolno,
   Endschool   : fivedigits;

   FirstRecord,
   result      : boolean;
   PageNmr,
   LineNmr     : positive;
   spacescount : 0..136;

   leafile,
   schfile,
   sbjfile     : Descriptor;
   res         : FileStatus;

{*}
PROCEDURE pntrycnd*;
BEGIN (* pntrycnd*)
WITH cnd DO BEGIN
   result := true;
   freadindxfa, cnd, sizeof(cnd), RecordNo);
   IF NOT deleted THEN BEGIN
      IF FirstRecord THEN BEGIN
         FirstRecord := false;
         Getsch(schfile, schoolnmr, sch, result2);
         FOR j := 1 TO 2 DO leacode[j] := schoolnmr[j];
         Getlea(leafile, leacode, lea, result);
         pageheader;
      END ELSE
         IF schoolnmr <> prevschool THEN BEGIN
            IF schoolnmr > Endschool THEN result := false
            ELSE BEGIN
               Getsch(schfile, schoolnmr, sch, result2);
               FOR j := 1 TO 2 DO leacode[j] := schoolnmr[j];
               Getlea(leafile, leacode, lea, result);
               pageheader;
            END;
         END ELSE
            IF LineNmr + nersubjects > Pagesize THEN pageheader;
      IF result THEN BEGIN
         prevschool := schoolnmr;
         GetAge;
         PrintCand;
      END;
   END; (* IF NOT deleted *)
END; (* WITH cnd *)
END; (* pntrycnd*)

BEGIN (* NTRYSCHS *)
   Openbtr(SKEW);

   reset(infile, SKEBYEAR);
   read(infile, thisyear);
   close(infile);

   openr(LEAREL, LEANDX, leafile, R, res);
   result := true;
   REPEAT
      prompt('Start school number? '); readln(schoolno);
      prompt('End school number? -1 '); readln(Endschool);

      FOR j := 1 TO 2 DO leacode[j] := schoolno[j];
      Getlea(leafile, leacode, lea, result);

      StartKey := hashcnd2(lea.foreign, schoolno, '001');
      IF StartKey = minint THEN result := false;

      IF alldigits5(Endschool) <> 0 THEN result := false;

      IF NOT result THEN writeln('ERROR: invalid data entered');
   UNTIL result;

   prompt('Start page number?: '); readln(PageNmr);

   prompt('CONFIRM (Y/N): '); readln(reply);

   IF (reply = 'Y') OR (reply = 'y') THEN BEGIN
      FirstRecord := true;

      openr(SCHREL, SCHNDX, schfile, R, res);
      openr(SBJREL, SBJNDX, sbjfile, R, res);
      rewrite(outfile, LP3);

      DoPart(CANDREL, CND2NDX, R, StartKey, pntrycnd, result);

      writeln;
      writeln('Processing complete - output in spoolfile LP3');
      writeln;
      writeln('Last school printed: ', prevschool);
      writeln;
      writeln('Last page printed: ', PageNmr - 1);

      closer(schfile, SAVE);
      closer(sbjfile, SAVE);
      closer(leafile, SAVE);
      close(outfile, SAVE);
   END ELSE writeln('NOT CONFIRMED: PROGRAM ABORTED');
END. (* NTRYSCHS *)
```

6.6.3 Comments on the tests -

The application systems described in this section are in no way intended as a direct comparison between DBaseII and ADIM. The Commodities Buyer Agency system uses DBaseII, and since DBaseII has become a very popular system, I think that a description of the commodities system helped to place ADIM's capabilities into context. It should also be said that this work was done early in the design and implementation of ADIM, circa 1981. It is also interesting to note how quickly DBaseII acquired a very wide user community. I believe that this is more an indication of the need for personal information systems than of the quality/capabilities of DBaseII. This point is illustrated by the Commodities Buyer Agency example.

Simulation and theoretical work have been done to estimate the behaviour of B-trees [GUDES, YAO, QUITZOW]. Although, this type of work can provide good analytical results, in the final instance, practical issues will determine the performance of a particular implementation. A case to illustrate this point, is DBaseII's implementation of B-trees. In practical terms, it would not matter if DBaseII's indexes were implemented as ISAM directories. A deletion of an item in DBaseII does not erase the item, it only marks the item as deleted. One needs to 'purge' and re-organize the indexes to actually

delete items.

Against this background, the Examination Monitoring system put the core of ADIM to the test. This system completely proved the feasability of B-trees, as implemented in ADIM. It produced performance figures many times better than DBaseII could have produced (projected figures), had DBaseII been capable of handling files of the magnitude required by this application. It should also be noticed that in the implementation of the Examination Monitoring system some inefficiency was introduced by coding ADIM-File Manipulation Language (FML) in Pascal, rather than 'C'. The inefficiency is due to the strong data typing of Pascal, which makes data type coersions cumbersome to implement. Unfortunately, this type of 'dirty' programming is required at the lowest level of file systems such as FML.

6.7 Cost estimation -

The complexity analysis of algebra operators helps in establishing very general ideas about the time required to evaluate a particular expression, but it is limited to using little information about the relations involved except their cardinality. On the other hand, general cost functions are difficult to establish, but in a

particular environment they can provide us with a good deal more information so that a decision regarding specific strategies for evaluation can be adopted in the processing of certain queries.

Decisions about a strategy for processing a query in ADIM are made in two different places: the ALFRED-VC to K translator, and in ALFRED-K's virtual machine, otherwise known as a P-unit.

A global strategy for the evaluation of a user's query is chosen by the C-unit. The decision is made by the application of general principles of relational optimization. The transformation of algebraic expressions into equivalent and generally more efficient expressions is governed by a set of rules. The rules are derived from techniques proposed by Pecherer [PECHERER] and Palermo [PALEREMO]. Also, rules to deal with cases due to the use of the decomposition process, are included. More details about the decomposition process and general rules of optimization in ADIM, can be found in Chapter 5.

In this section, a second form of "optimization" of query expressions is discussed. To find equivalent expression, i.e. one which takes minimal time to evaluate, for a given relational expression, it could

easily take longer than the actual time needed to evaluate the original expression, itself. Hence, ADIM does not attempt true optimization, but instead, tries to quickly determine a good equivalent expression. A good equivalent relational expression in this context means an expression equivalent to the user's query expression, which can be evaluated in a time close to the optimal equivalent expression, if there was one. The choice of a good equivalent expression is based on a cost analysis of a number of equivalent expressions.

Queries received by a P-unit are again transformed by the application of transformation rules. This time, ADIM only uses a small number of the rules proposed by Palermo and Pecherer. The aim is to reduce the number of alternative evaluations for a given query to only a few cases. This assumes that some global optimization of the query has already taken place [See Chapter 5]. In this way, we only need to concentrate on a few significant cases. In fact, the rules used at this stage only involve the operators restriction and/or projection, plus one more operator. In ADIM, projection is normally evaluated concurrently with other operators. Hence, we can think of the cost to evaluate projection as being zero, except of course, when projection is the only operator to evaluate. Also, by using the parameters of B-trees, we can estimate with certain accuracy the volume

of data involved in a restriction operation. The rules for restriction seek to benefit from this information.

Given a number of different alternatives for the evaluation of a query, a decision is made by estimating the total cost for each alternative, and then, choosing the one with the lowest cost.

To estimate the total cost for a given relational expression, an evaluation tree is built for the expression and cost is allocated to each node of the tree. Internal and external nodes are costed. The cost analysis assumes that basic relations and intermediate relations which are created during the evaluation of the tree, are already sorted into their correct retrieval key. This is a realistic assumption. Basic relations are often accessed by their key, and therefore, they can be considered as sorted. In the case of basic relations being accessed by non-key attributes, sorting can be added as another leaf to the evaluation tree. Let us examine now the case of intermediate relations.

Consider an evaluation tree for a given query. For each internal node in the tree, we know before we evaluate the node which specific attributes will be needed in its evaluation. Hence, we can create the intermediate relations with their keys sorted in the

correct order, i.e. we build the key for an intermediate relation with the attributes by the evaluation of the parent node of the relation.

It should be noticed that the only relations that might need to be sorted are basic relations. Because of decomposition, these relations are bound to be small and therefore, their sorting would not add significantly to the evaluation of the query expression.

Now to estimate costs, we need to define our cost unit. In conventional computers, the time needed to move one word from disc into main memory, is most likely to be the same as the time needed to move a whole physical block of the disc into memory. Because of this, ADIM defines the size of a B-tree page as a multiple of physical blocks in a disc. Thus, it makes sense then, to define as our cost unit, the block. For the purpose of cost analysis, we can equal one B-tree page to one block. Notice that in ADIM, as well as in any other data base management systems, data traffic is the factor that determines the speed of the system, overall.

The total estimated cost for an evaluation tree is defined by the sum of the estimated cost for each of the nodes in the tree. The basic relations, represented by the external nodes (leaves) of the tree, are assigned a

cost equal to the number of blocks occupied by their B-tree. The cost of internal nodes, the operators, depends on the cost allocated to their children and the operator represented by the node itself.

The creation of intermediate relations is not costed, since their contribution to the total time taken by the evaluation of the whole tree, is assumed to be proportioned to the evaluation of internal nodes of the tree.

Before proceeding with the discussion on how to determine the cost for each node in an evaluation tree, let me introduce some notation:

N_R = number of blocks used by the B-tree for relation R;

T_R = number of tuples in relation R;

n_R = degree of B-tree for relation R ($2*n_R$ records can be stored in one B-tree page);

K_R = occupancy factor for relation R, i.e. $(N_R*2*n_R)/T_R$;

C_X = cost in blocks (to be retrieved from disc) to evaluate node X of the tree. It is assumed a fixed time is needed to move one block.

Now, I shall proceed to explain the cost allocation schemes for each of the possible elements to be found in an evaluation tree.

BASIC RELATION -

The cost associated with a given relation R is determined by the number of blocks, N_R, in the B-tree for R. The value of N_R can be determined in three different ways. The first and most obvious way, is used when the B-tree is stored in one of ADIM's devices. The device descriptor holds this information. The second way of determining the value of N_R, is used when the B-tree for R has been constructed as a file in the host operating system. The value of N_R is defined by the ratio of the length of the file and the size of a page in the B-tree. The third and final case, occurs when the two previous methods fail. In this case, ADIM's functions 'create page' and 'destroy page' must keep count of the number N_R. This is in fact simpler than to keep than the count for the number of tuples in a relation, T_R. We denote the cost associated with a given relation R, by:

$$C_R = N_R$$

JOIN, UNION, DIFFERENCE AND INTERSECTION -

These four operators are evaluated in a similar manner. Consider the two relations R' and R" and one operator, say x. The application of operator x to relations R' and R", produces the relation R. The algorithm to evaluate x, assumes the relations R' and R" are sorted on a common list of attributes A, R'[A] and R"[A], respectively. The main part of the algorithm is presented below:

.
.
.

Step I+1.

a. If r'[A] = r"[A] then for

(i) R':*:R", build tuple r for R from r' and r";
(ii) R':.:R", build tuple r for R from r';
(iii) R':+:R", build tuple r for R from r';
(iv) else, do nothing.

b. Read next tuple r' from R; and next tuple r" from R".

Step I+2.

a. If r'[A]>r"[A] then for

(i) R':+:R", build tuple r for R from r";
(ii) else, do nothing.

b. Read next tuple r" from R".

Step I+3.

a. If r'[A]<r"[A] then for

(i) R':+:R", build tuple r for R from r';
(ii) R':-:R", build tuple r for R from r';
(iii) else, do nothing.

b. Read next tuple r' from R'.

Step I+4.

Iterate steps (I+1)-(I+3), until ... END.

From the given algorithm we can then deduce the following:

$$C{:}+{:} = N_{R'} + N_{R''}$$

$$C{:}.{:} = N_{R'} + N_{R''}$$

$$C{:}-{:} = N_{R'} + N_{R''}$$

$$C{:}*{:} = N_{R'} + N_{R''}$$

The cost estimation functions given above, assume a memory management based on a stack. Once a page of a B-tree is pushed into the stack, it stays there until all of the tuples in the page have been processed. This is naturally enforced by the algorithm being discussed, since ADIM's stack has capacity to store several pages.

PRODUCT -

This operator is evaluated by iteration over one relation, R", for each tuple in relation R'. Thus, the estimated cost for the evaluation of R=R'(*)R", is given by the formula:

$$C_{(*)} = N_{R'} * N_{R"}$$

PROJECTION -

Normally, projection is evaluated in conjunction with another operator. Hence, the cost normally estimated for projection is nil. In the unlikely case that projection is evaluated on its own, the cost allocated to it is N_R since R is assumed to be sorted. Thus, the cost function for projection is:

$C_\wedge = \emptyset$, if evaluated in conjunction with any other operator.
$C_\wedge = N_R$, if evaluated on its own.

RESTRICTION -

Perhaps the most interesting cost function is the one for restriction. It is an important operator because of the frequency with which it appears in queries. Often, it is also used to derive new algebra operators. Examples of this, are: a generalized join and vector type operators.

Because of the above reasons, ADIM uses a cost function for restriction, which is much more refined than any other cost function, previously discussed. Also, ADIM chooses rules of optimization involving restriction, in preference to other rules.

In Chapter 5, we learnt that restriction's conditions in ALFRED-K are expressed in 'clausal' form, i.e. a search condition $Q=[q_1$ and q_2 and ... and $q_n]$ on a relation R is always expressed in conjunctive normal form. The conditions q_i are, in turn, lists of disjunctions. Because of this, ADIM searches the list Q of conjunctions, first of all, for a list q_i of disjunctions which includes an appropriate condition on the key(s) for R. For example, in the query:

RETRIEVE students WHEN [name = "Jones" and ...?

which can be expressed in ALFRED-K, by:

```
students @ [[:([name = "Jones"],[]), ...]]
```

the condition expressed by the clause :([name = "Jones"], []) will be selected by ADIM, to help in the determination of cost estimates.

If no clause q_i meets the above requirements, then there is not much that ADIM can do, and hence, the cost function for the restriction is taken to be:

$$C_{@} = N_R$$

In fact, this situation is unusual, since relations are normally sorted on the correct key, as it was seen earlier on in this section. In any other case, relations can always be sorted previous to the evaluation of the restriction.

Now, let us consider the normal situation, i.e. a suitable condition q_i on a relation which is sorted on the correct key. In a restriction of this type, ADIM distinguishes three uses:

(i)	equality	(=)
(ii)	less	(<, <=)
(iii)	greater	(>, >=)

Cases (ii) and (iii) are symmetric, and therefore, conclusions for case (iii) are identical to those for case (ii). Because of this, I will only discuss cases (i) and (ii).

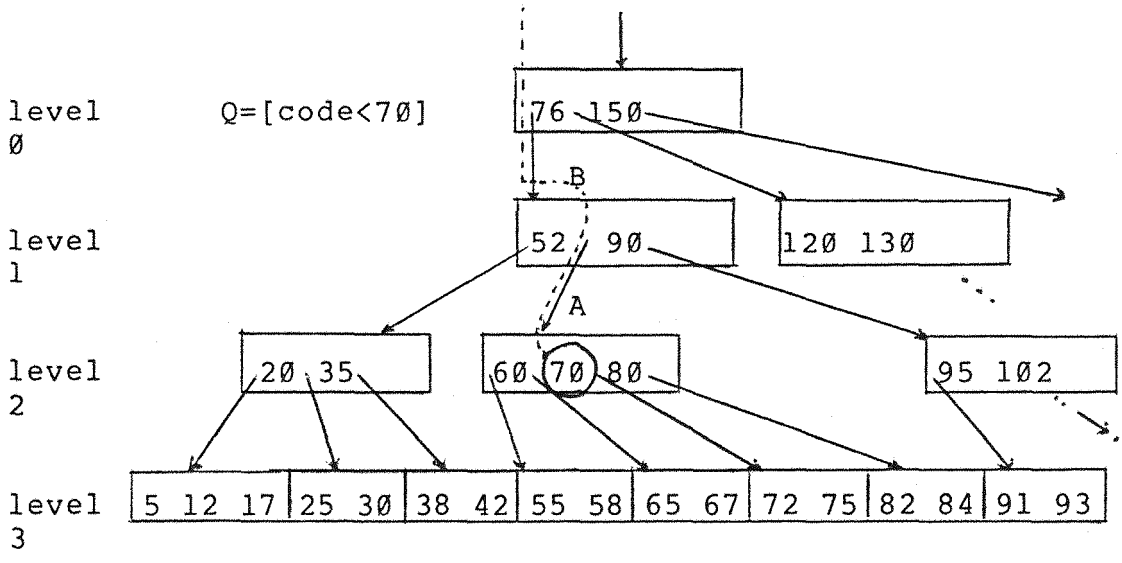
Case (i) is perfectly straightforward. The maximum number of pages to visit, is determined by the height of the B-tree for the relation. This is never a large number. In practical cases, even for large relations, this number is unlikely to be more than half a dozen pages. Because of the very high probability of finding the sought item, near to the bottom of the B-tree, ADIM defines the cost function for this case, to be equal to the height of the B-tree, i.e.:

$C_{@} = h$, where h is the height of the B-tree

The accuracy of the estimated cost in case (ii) is certainly, more important than in case (i). A retrieval by range may access a very large number of pages, since at least, a partial traversal of the B-tree for the input relation R, will be necessary. Also, the size of the B-tree for the result relation depends on the number of pages retrieved from R.

The study of case (ii) can again be divided into two categories. I shall call these categories: restriction

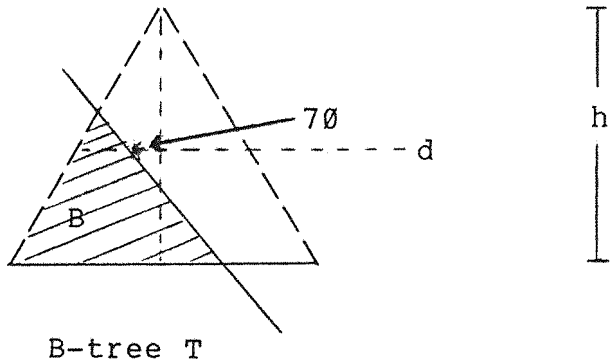
type L and restriction type R. Let me explain these categories. Consider the B-tree T and condition Q, below:



B-tree T (h=4)

Figure 6.12

First, an explanation for restriction type L. Consider the key for the B-tree, to be the attribute code. We search the tree T for the first item, such that code<=70, even when Q=[code<70]. I have marked with a broken line the walk down the tree to item with code=70. Notice that every item on the left of the broken line was code<70. Hence, the name 'restriction type L'. Now, take any sub-tree with root on the left of the item which code is 70, in page A. Call this sub-tree S. Every item in sub-tree S also has code less than 70. The situation as described so far, can be depicted by:



B-tree T

Figure 6.13

In figure 6.13, the tree T is represented by a triangle. Every item in the darkened area of the triangle has code<7Ø. Similarly, every item outside the darkened area B, has code>=7Ø. Hence, the cost of evaluating a restriction of type L, can be determined by a calculation of the number of blocks in the darkened area B of the triangle.

To explain a restriction type R, consider again the tree T of the previous example. This time, the condition is Q'=[code>=7Ø]. The area for the qualifying items appears now on the right hand side sector of the triangle:

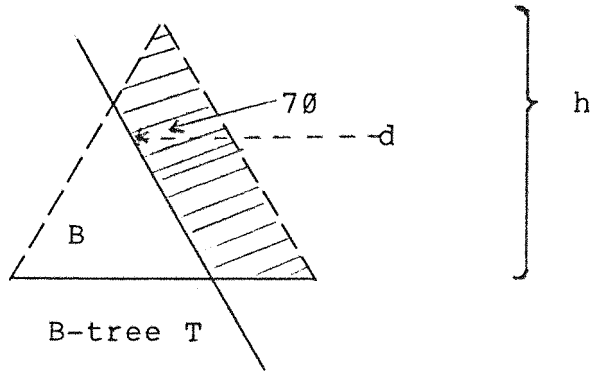


Figure 6.14

Do you remember the symmetry of cases (ii) and (iii)? To determine the cost of a restriction type R, calculate the area in blocks for the whole of triangle T, and also the area of triangle B. The cost for this type of restriction is then, given by the difference between the area of T and the area of B. The number of blocks in B are calculated by using the negation of the original condition Q', e.g. in our example: Q"=not Q'=[code<7Ø].

But, how is the number of blocks in area B calculated? First, allow me to answer a simpler question: how many blocks does a B-tree X have?

The maximum number of pages (blocks) that a B-tree can have is determined by its degree and height, denoted by n and h, respectively. A formula to calculate this maximum is given below:

$$\text{Max. No. of Pages} = \frac{(2*n + 1)^{h} - 1}{2*n} \qquad (1)$$

Now, if we know the occupancy factor of the B-tree for relation R, denoted by K_R, we can estimate more accurately the number of pages N_R, in the B-tree. From (1) above follows:

$$N_R = \frac{(K_R * 2 * n_R + 1)^h - 1}{K_R * 2 * n_R} \qquad (2)$$

Simulation studies and practical experimentation with B-trees [YAO, NAKAMURA, ROSENBERG] have demonstrated that a conservative figure for the occupancy factor in large B-trees, is around $K_R = Ø.7$. However, ADIM can determine with greater accuracy the value K_R, for any given relation R. The formula used by ADIM to calculate the occupancy factor, is:

$$K_R = \frac{N_R * 2 * n_R}{T_R} \qquad (3)$$

As it was explained earlier on, the value N_R is easily obtainable in ADIM. The same is also true for T_R and n_R. In Section 6.5, we discussed the descriptor for each open relation. This descriptor stores the values T_R and n_R, in the fields reldum.reltup and reldum.n, respectively.

Let us assume that the value K_R is also valid for

every sub-tree in the B-tree. This is not an unrealistic assumption, considering the uniform distribution of data enforced by the overflow and underflow algorithms used by ADIM [QUITZOW]. By using formula (2) above, we could calculate the number of blocks in any given sub-tree, if we knew the height of the sub-tree.

In our example in order to get to page A, we walked down to level d of B-tree T. We know it is level d, because in our way down the tree T, we visited d+1 pages. Now, let us assume that we also know the height h of tree T. Then, we can deduce the height of any sub-tree which root page is pointed by a pointer in page A. This height is determined by the difference: h-(d+1). See digure 6.15, below:

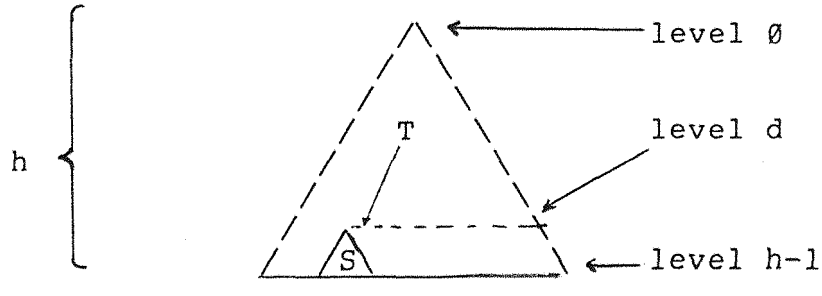


Figure 6.15

The assumption about us knowing the value of h, is a fact in ADIM. As soon as a relation is opened in ADIM, its height h is determined. Also, while the relation R remains opened, the value of h is updated whenever the

tree shrinks or grows.

In page A of our example there are q=3 items. The item with code=7Ø is in position, i=1. All of the items stored in each of the sub-trees on the left of item code=7Ø, satisfy the condition Q=[code<7Ø]. Because of this, we referred to them as the qualifying sub-trees. In page A there are: q-i=2, qualifying sub-trees. See figure 6.16, below:

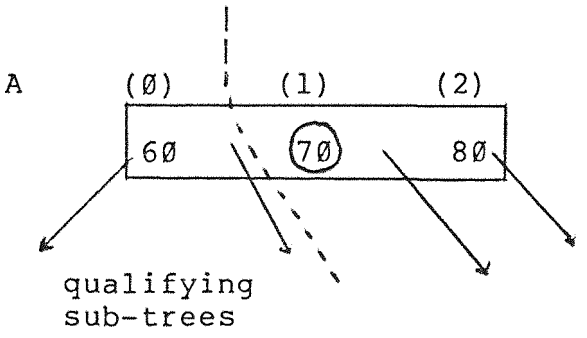


Figure 6.16

From formula (2), we can now derive a formula to estimate the cost associated with level d, $C_{@:d}$:

$$C_{@:d} = (q_d - i_d) * \frac{(K_R * 2 * n_R + 1)^{h-d-1} - 1}{K_R * 2 * n_R} \quad (4)$$

Let us now look at page B, the ancestor of page A [Figure 6.12]. The sub-trees to the left of the element immediately to the left of the broken line, are all qualifying sub-trees. By using the same analysis that we

used for page A, we can estimate the cost associated with level d-1. In this manner, climbing up the B-tree while re-tracing our steps, we calculate the cost at every level until we reach the root page. Thus, the final cost function for restriction type L is:

$$C_{@} = \sum_{j=\emptyset}^{d} C_{@:j} + d \qquad (5)$$

where

$$C_{@:j} = (q_j - i_j) * \frac{(K_R * 2 * n_R + 1)^{h-j-1} - 1}{K_R * 2 * n_R}$$

and

$$K_R = (N_R * 2 * n_R)/T_R$$

It also follows from this analysis, a cost function for restrictions type R. The formula is:

$$C_{@} = N_R - \left(\sum_{j=\emptyset}^{d} C_{@:j} + d \right)$$

The values for $C_{@:j}$ have been obtained by using the negation of the original condition Q.

One minor point. The count of levels d in the analysis above, is not obtained by directly counting the number of levels descended, as suggested in the discussion. During searches, ADIM pushes every new page

into a stack, and on backtracking pop them out. The effect of this is that during the walk down the B-tree, its pages are stacked up, and during the climbing up, these pages are thrown away. Thus, if we record the position in the stack for the root page of the B-tree, we can always establish the current level in the B-tree. See Figure 6.17.

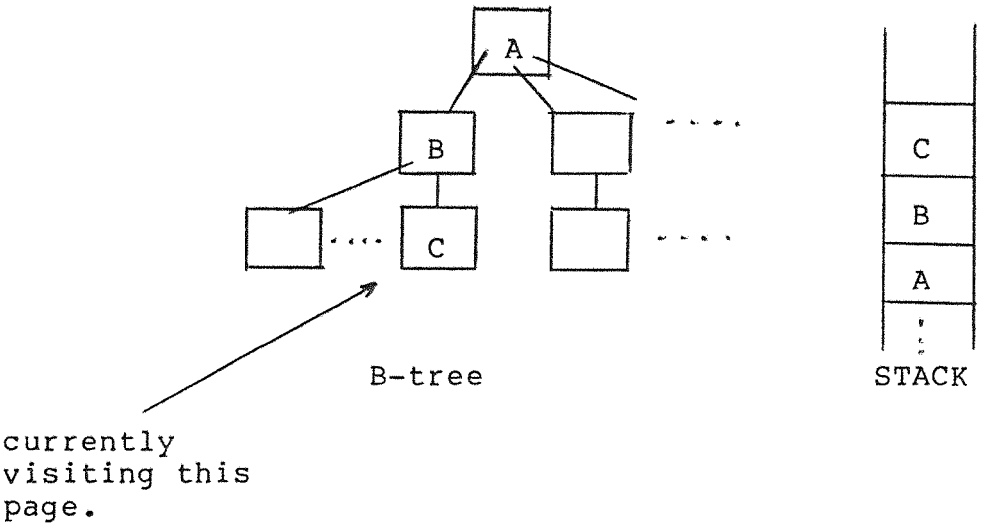


Figure 6.17

Also, because of the stack, once we calculate $C_{@:d}$ for page A, in Figure 6.11, we do not need to get page B from disc again, since it already is in the stack. This explains why, we only add d in the calculation of $C_{@}$.

Finally, it should be noticed that the cost function for restriction is of special significance, since restriction is also used in the definition of all the vector aggregate operators, and a generalized join

operator.

6.8 Dynamic Structures -

It has been suggested that dynamic access methods particularly trees [KNUTH] and B-trees [COMMER79], etc, may be troublesome [HS75] as a storage structure for files on paged secondary storage devices. As a consequence there exists a widespread belief that implementation of B-trees in a relational environment may incur a performance penalty when compared to other schemes for the management of large volumes of data [HS75, HELD75].

It was precisely these views and opinions about the inefficiency of B-trees that led me to a more detailed study of them. My opinion is that B-trees may be inefficient where they have been implemented on top of the existing file structure of the host operating system. Normally, this file structure is of a static type, supporting sequential files and/or static directories, e.g. UNIX [RT74] and CP/M [CPM]. This way of implementing B-trees [ASHTON, MISTRES] is bounded to be inefficient. Firstly, the dynamic re-structuring of B-trees conflicts with the static files of the operating system. Secondly, the number of re-directions needed in a search of a B-tree are multiplied many times over by

the re-directions imposed by the file system of the operating system. For instance, consider the case of a B-tree with a height of five nodes for the keys and implemented on top of the UNIX file system. Since UNIX normally imposes three re-directions in big files (like the one in the example), in order to access one item in the leaves of the B-tree, fifteen pages of data will have to be examined.

Clearly, situations such as the one in the above example are not desirable in a relational system where associative searches of the data space may cause extensive examination of secondary memories. As an alternative, I decided to explore a situation where B-trees are implemented as hardware devices, so by-passing the file structure of the host operating system and its inherent inefficiencies.

For retrieving data by equality, an access method based on a carefully designed hash function will certainly be extremely difficult to beat in performance, but notice that the same hash access method will be disastrous for a retrieval by range [see sections 6.2 and 6.3]. A retrieval by range on a hashed key of the relation will force a sequential search visiting every tuple of the relation in question. On the other hand, given a stable relation, static directories such as the

ones used in INGRES [SWKH76] will improve considerably the performance for the retrieval by range case while still behaving moderately well in the retrieval by equality case. However, when confronted with volatile relations, i.e. relations subject to continuous up-dates, deletions and additions, they are no solution. This is due firstly, to an excessive number of overflow pages generated by partial reorganizations of files between up-dates, secondly to the need to search sometimes sizeable sequential files created by delayed updates,and thirdly to the relatively high cost of the periodical reorganization of those files supporting the relations affected.

From the discussion above, B-trees as candidates for the unique file structure of ADIM, meet conditions 2, 3, 4 and 5. Condition 6 is also fulfilled by B-trees, as demonstrated in Section 6.7. However, condition 1 remains for closer scrutiny. Obviously, this condition is not fully met by B-trees. Nevertheless, by using a memory management system based on a stack discipline (LIFO), a whole branch of a B-tree can be loaded into main memory, so reducing the access factor to one, for all successive pages after the first page of a range (>, <, etc) retrieval. Meanwhile, equality retrievals and the first page of a range retrieval have an access factor upper bounded by the height of the B-tree.

It is precisely, on volatile data bases such as the one used in personal systems (banking, home management, office automatization, etc) that B-trees as a particular case of dynamic data structure have the potential to provide major gains in performance. Reorganization of data on the fly as a central characteristic of B-trees does avoid all of the perils of delayed updates, i.e. overflow pages, huge sequential files and expensive periodical reorganizations. I am convinced that in a stable environment B-trees also perform better than many other data structures.

By choosing B-trees as the file structure for ADIM, all six conditions of section 6.1 can be met. Condition 1 to 5 can be met fully, and condition 6 partially. Because of this, and in preference to many other file structures (randomizing directories included), I believe that the use of B-trees is highly advantageous. This is demonstrated by the empirical tests in section 6.6. Consequentially, ADIM uses a unified file structure based on B-trees. I did not experience any major problem in the implementation of them and I can also produce good reasons for their use.

CHAPTER 7

IMPLEMENTATION OVERVIEW

7.1 Introduction -

In order to avoid unnecessary complexity in the exposition, I will concentrate only on some aspects of the implementation of ADIM. Thus, I will cover the core of the ADIM system and those parts which provide a focus of interest for implementators using ADIM in future applications. For these reasons then, the discussion is centred around the implementation of a P-unit.

It should be noticed that the implementation of C and G units as well as some aspects of the P-unit have already been covered in chapters 4, 5 and 6.

The description of the implementation is broken down into six areas: i) sub-systems of ADIM as invoked by users; ii) the Compiler Query Language a virtual machine

for ALFRED-K; iii) a file manipulation language for the CQL; iv) utilities; v) some special files; and vi) system catalogues. A detailed discussion of these six areas follows.

7.2 Sub-systems -

In this section, the implementation of three sub-systems is examined: ALFRED, dbmk and mkdev. They are not the only sub-systems of ADIM, but they are representatives of the implementation problems in sub-systems of their type.

7.2.1 ALFRED -

The ALFRED sub-system is entirely written in PROLOG. It is normally used as a G-unit, but the data base administrator can also use it as a front-end to the C-unit. The ALFRED sub-system has three parts: the parser and lexical analyzer which recognizes valid sentences; the decomposition part which breaks down the queries into queries involving only elementary relations; and the code generator which has two passes, the first of which creates unique names and sets up necessary tables, and the second pass which issues a function call in

ALFRED-K form for every query involving elementary relations. The output of ALFRED can be compiled and executed directly (if the C-unit is present) or it can be a file containing the queries in ALFRED-K form (Chapter 3) for a delayed execution.

7.2.2 dbmk -

This is for creation of new data bases. It creates the system relations: 'relation' and 'attribute'. It also makes entries in the sequential file 'alldbs'. This sub-system is written in the language 'C'.

By invocation of the function existdb(), it checks if the named data base has already been created. It also checks if the specified device exists within the system. Once that the above tests have delivered a positive result, dbmk proceeds to create the data base by obtaining space for the relation. Finally dbmk records the existence of the new data base in the file 'alldbs' of the host operating system.

It should be noticed that relations in a data base are described in terms of relations. These are the relations: 'relation' and 'attribute'. These relations are in turn, described by themselves, so permitting the

shared use of software for the manipulation of catalogues belonging to the system and relations belonging to users. Since at the time of invocation of dbmk, the catalogues for the new data base do not exist, it is necessary to maintain the correspondence between the sizes of the 'C' structures for the catalogues and the sizes given by ADIM to the same catalogues. This problem only arises when ADIM is ported to a new operating system. For this reason and to improve portability, dbmk makes extensive use of the function pointer() which takes care of variations in the data types of 'C'.

7.2.3 mkdev -

The purpose of mkdev is to create an environment for ADIM independent of the peculiarities of physical devices. Thus, an ADIM device could correspond to a sequential file in a given operating system or it could be a magnetic disc or any other physical device used as secondary memory. Once mkdev has run, the relevant entry in the local file 'alldbs' will be established as a record of the relationship between the device (or file) in the host machine and a device name within the ADIM system. Notice that mkdev tests for the existence of the device before creating it.

A device in ADIM consists of map pages and data pages. The map pages are used to maintain a bit map of the data pages. Data pages are used to store relations.

In order to implement mkdev, the following functions were also implemented: opendev(), to open a device; closedev(), to close a device; zeromap() and maper(), to mark a data page in use within the device; unmask(), to free a data page; mask(), to do the bit mapping. References to some of these functions will be made again in section 7.4.

7.3 CQL -

A description of the implementation of the CQL follows. These functions are invoked directly by the application(s) using a P-unit or by any G-unit (including an ALFRED sub-system).

7.3.1 append -

This function appends a tuple to a named relation. It first tests the existence of the relation. Then, it prompts the user with the names of each attribute, and waits for input. The append() function makes extensive

validations of input data. For this purpose, append() invokes utility routines which are discussed in section 7.5. Finally, append() handles the new tuple to the function insert() of the FML (section 7.4) for addition to the named relation.

7.3.2 display -

This function prints the named relation in the user's terminal. The implementation of display() is a rudimentary application of a generalized mechanism for building report generators. This mechanism is based on a table with five columns. The first column contains the name of the function invoked, in this case 'display'. The second column contains the name of a function which produces the headings for the report. The third column contains the name of the function to print individual tuples. The fourth column contains the name of the function which handles the 'end-of-tuple' delimiter. The fifth and final column contains the name of the function which handles the printing after the last tuple has been printed.

In the case of display(), the columns are as follows:

1. display - the name of the command.

2. printhead - prints the names of the relation and the attributes.

3. printtup - prints the tuple. In turn, this function invokes printatt() which prints every value per attribute, using the corresponding format, i.e. it prints an integer as an integer and not as a string of characters.

4. preol - invoked after the last attribute/value for the tuple has been printed. This prints a vertical bar (|), followed by the characters 'LF-RETURN'.

5. preor - prints a horizontal line and two 'LF-RETURN'.

I would like to stress that I have concentrated in providing a general mechanism for the preparation of reports. The display() function is only a trivial example of the use of this mechanism.

7.3.3 create -

The invocation of this function creates a new relation. Firstly, it interactively collects information about the name of the relation, the device where it will be created, the name of the attributes and their format. Once this information is collected, it proceeds to validate the names, formats and devices. Sometimes, the device is unknown to ADIM, a relation with such a name

already exists, etc. After the validation stage, information on keys for the relation is collected. At this point, it supplies the user with help to set up the primary key and in some cases it does it for him/her. Error recovery is graceful.

7.3.4 join, project, union, select, ... -

These are the functions which implement the query sub-language. Typically, they will:

i) open the source relation(s);

ii) create an empty relation for the result. If this relation is temporary it can sometimes be maintained in buffers in main memory, so speeding up execution.

iii) the algebra operation is performed and the generated tuples are stored in the relation created in step (ii).

Step (iii) is perhaps the most interesting. Depending on the boolean condition in operations such as join() and select(), partial traversals of the B-trees are attempted. In some other cases, tuples are obtained with one invocation of gettuple(). If all of this fails, then a complete traversal of the B-tree is performed.

7.3.5 Remarks -

It should be noticed that throughout this stage of the implementation, references to relations are immediately transformed to a descriptor. A descriptor is an in-core summary of the details held about one relation in the system catalogues. This mechanism avoids the inefficient and often repeated consultation of system's catalogues held in secondary memory (which is considerably slower than main memory).

7.4 FML -

The File Manipulation Language (FML) is the interface between the CQL (section 7.3) and the operating system / host computer. It is a layer of safety, to ensure portability of ADIM. A list of the main functions and a brief description of their implementation follows.

7.4.1 closer -

This function releases the descriptor of an open relation. It is the counterpart of openr(), below.

7.4.2 openr -

This function consults the system catalogues and creates an in-core summary of the characteristics of the named relation. For this, it needs to open the relation 'relation' and the relation 'attribute'. Unfortunately, to open these relations a descriptor for them is required. Hence, the functions reldesc() and attdesc() were provided. These functions "hand-craft" the descriptors for 'relation' and 'attribute'. A locking control for devices is also activated in certain cases, by the invocation of openr().

7.4.3 Increate -

Similarly to create in CQL, it creates a new relation. This function is used to create a relation where details about the relation's name, the names of the attributes, the format of the attributes and the key are implicit in the query. For instance, the result relation in a join or project. In order to gather information from the source relations, it uses the functions get_atts() and pull_att(). The first of these functions normally invokes the second, which collects information about one particular attribute in a relation.

7.4.4 insert, search, delete, travertree and partial -

These functions are a recursive implementation of what their names suggest. Thus:

i) partial(), is a partial traversal of the B-tree for a given relation;

ii) travertree(), is a full traversal of the tree;

iii) insert(), appends a new entry to the tree;

iv) search(), finds an entry in the tree; and

v) delete(), deletes the keyed entry from the tree.

These functions need the descriptor for the given relation. This is normally provided by openr(), together with the searching keys.

I feel that the implementation of these functions is highly compact. This makes possible the running of ADIM in small systems, typically, a CP/M based system or a small configuration of UNIX.

Perhaps the most interesting aspect of this implementation is the flexibility built into these functions. At least, one parameter in each of these functions accepts the name of another function. Thus, for instance, a trivial implementation of join() could have been:

```
travertree(descr1, ..., travertree, param2);
```

where,

```
struct param2 {
        descriptor descr2;
        condition join_cond;
} param2;
```

This mechanism is used often in the implementation of ADIM. In particular, in the case of the FML implementation, it provided me with a powerful and simple method to implement composition of functions.

7.5 Utilities -

For the purpose of this explanation, I have grouped the utilities into seven groups.

7.5.1 Memory management -

These functions implement a stack discipline for the management of memory. No other type of memory management is required to handle queries. This discipline is extremely well suited for ADIM, since the relations are

stored as B-trees. This combination makes a 'garbage collector' absolutely unnecessary.

Not only the software to write was reduced, but also, the stack discipline provides a natural 'cache memory' for ADIM. As an example consider, the previous trivial join. Pages grabbed by the first invocation of travertree() are only released once the second invocation (the parameter to the first) of travertree() has fully finished with them.

A simpler example is provided by the query:

```
RETRIEVE employee WHEN
        salary > 10K ... ?
```

Here, a partial search of the B-tree loads and unloads pages in main memory until the first qualifying tuple is found. From this point onwards, all of the tuples to the right of this tuple (in the page) as well as all the pages in the sub-tree below, qualify. Because of this, the whole of the qualifying sub-tree can be further processed by stacking its pages and then poping one page at a time for processing. Notice that once the first tuple is found, no more testing of the qualification is necessary.

In the case of the join example:

```
travertree(....,travertree, ...);
```

the stack naturally handles backtracking.

In the scheme of memory management described, the most important functions are:

salloc() - grabs a page from the stack;
sfree() - releases the page.

7.5.2 Descriptors -

The functions reldesc() and attdesc() provide a facility for quick creation of a descriptor for 'relation' and 'attribute', respectively. These functions were originally implemented to bootstrap ADIM, so that the catalogues of the system could also be relations. To understand the problem, consider the insertion of the tuple containing information about the relation 'relation' in the relation 'relation'. To do this, it is necessary to invoke insert(), which needs as parameter a descriptor for the relation in which the tuple is going to be inserted. This descriptor is normally obtained by opening the named relation. Since

the relation 'relation' does not exist when we want to insert the tuple describing the relation 'relation' in the relation 'relation', we need to create a descriptor by different means. This is the purpose of reldesc() and attdesc(). These two functions also are an obvious short-cut to the catalogues of the system, which are consulted several times in the course of a query.

The function replica() makes a copy of a given descriptor. This is extremely useful when creating new empty relations out of old relations. A case of this is the result relation for a restriction operation.

7.5.3 Qualification -

Three functions were implemented to test tuples for qualification under operations requiring these tests. These functions are: compare(), nkcompare() and qualify().

The function compare() tests for equality, inequality or order, between two tuples belonging to relations not necessarily different. Keys are used by compare(), while nkcompare() is a version of compare() for those cases where searching keys are not available. The function qualify() is more suited for comparisons between a tuple

and a set of constants. Typically, compare() and nkcompare() are used in operations such as join, while qualify() is used by operations such as restriction.

7.5.4 Keys -

The function setkey() prepares a tuple image for searching in a given relation. This function sets the keys for searching. The counterpart to setkey() is clearkey(), which clears the searching keys.

7.5.5 Errors -

All errors and warnings are handled by the functions error() and warning(). They receive a set of parameters indicating position in the system, offending object identity and error class and type. Errors and warnings are classified according to the different sub-systems of ADIM. Furthermore, within a class they are also typified by another identification (number). This scheme of handling errors and warnings allows an ADIM system to maintain error messages and warning messages in relations like the ones used by other catalogues in the system. The advantage of doing this is twofold; firstly, by dynamic insertion of error and warning messages, an ADIM

system can be tailored to specific environments and applications; and secondly, a reduction of size of the ADIM system resident in main memory is achieved, since the messages which occupy considerable space are kept in secondary memory. In addition, I should mention that ADIM uses its own data base capabilities (retrieval, insertion, etc) to handle its error and warning messages. This makes the writing of special software for this purpose, absolutely unnecessary. Thus again, as implementator, I have benefited from the above scheme.

I feel that the described scheme for handling errors and warnings is a major contribution towards compactness in ADIM. In the previous paragraph, I have given one reason for it. A second reason, probably obvious at this point, is that software which is not written does not occupy any space. This is exactly what I have done here.

7.5.6 Strings -

A set of utilities to manipulate strings is an obvious need in any data base system. In particular, in the ADIM system, the following functions have been implemented and also made available for general applications:

cmp(s,t):	compares strings s and t;
strigth(s):	returns the length of string s;
strcp(s,t):	copies string t into s;
reverse(s):	reverses string s in place;
itoa(n,s):	converts the integer n into the string s;
concat(o,i1,i2):	concatenates strings i1 and i2 into string o;
indsex(tbl,entry):	find entry in sequential table tbl;
getline(s,lim):	gets line from tty into s and returns its length;
clean(s,sz):	cleans the string s of size sz;
move(r,a,sz):	moves the string a of size sz into r;
pad(a,sz):	pads the string a with blanks until size of a becomes sz.

7.5.7 Validation -

Functions to validate input data were implemented. They are available to applications as well. The functions are:

v_id(s):	validates the string s as an identifier;
v_form(s):	validates the format in the string s;
v_pint(s):	to validate the positive integer in s;
v_preal(s):	likewise, but for reals;

v_real(s):	validates real numbers;
v_int(s):	validates integers;
v_string(s):	validates strings;
v_char(c):	validates c as an ascii character.

7.6 Special Files -

This section discusses files of special significance in ADIM.

7.6.1 alldbs -

This is a sequential file (the only one) assumed to exist in the host operating system. This file is required to bootstrap an ADIM system. The contents of this file describe the devices available to ADIM and the data bases recognized by ADIM in a given computer.

7.6.2 devices -

The file 'alldbs' associates the names of devices and/or files in terms of the host operating system and the names of such devices and/or files in terms of ADIM.

7.6.3 FILES.h -

Tuning of ADIM is possible by changing the value of parameters defined in the files which names are post-fixed with .h. This follows the conventions of the UNIX operating system and the programming language 'C'.

7.6.4 IRC -

This is a shell or submit type of program generated by ALFRED and containing ALFRED-K expressions equivalent to the original ALFRED-U/VG query. This is normally used as an intermediate stage in the processing of ALFRED-U/VG queries.

7.7 System Catalogues -

The system catalogues for a given data base are kept in the relations: 'relation' and 'attribute'. Error and warning messages are kept in the relations: 'error' and 'warning'.

7.8 Some comments -

I feel that the implementation of ADIM fulfills the requirements for compactness, modularity and portability extremely well. The technique of using ADIM for its own implementation and maintenance, is to my belief, a major contributor to the above achievements. This is particularly true in the case of error and warning handling.

The marriage of B-trees and memory management based on a stack greatly simplified the implementation of the algebra operators. A 'garbage collector' is implicit in the above marriage: needless to say, the relevance to costing of queries, which is discussed more extensively elsewhere in this thesis.

CHAPTER 8

CONCLUSIONS AND FURTHER WORK

I have designed a "desk-top" information system which complies with the requirements of flexibility, portability, expandability and ease of use, demanded by personal systems. In designing such a system, I have found that efficiency of operation is the outstanding obstacle to its construction. I have undertaken a study of the problems of efficiency arising in the operation of such a system and provided an integral solution.

A high degree of compactness in the implementation of ADIM was attained by a careful selection of component parts. This selection of modules aimed for a minimalization of components to fulfil the requirements of ADIM. Alternatively, I could have chosen to offer users of ADIM a variety of good solutions to the problems posed by the design and implementation of each module of ADIM. This latter approach has already been tried in the

design of some relational data base management system [HUTT78, SWKH76] with a resulting product that it is too large and complicated for use as a personal data base management system. Let alone, an integrated information system, as described in this thesis.

ADIM assumes a small cardinality and degree in the relations of a data base. This state of the data base is attained by decomposition techniques applied to views, (Chapter 5). Thus, typically in ADIM, a query once parsed will refer to many small relations rather than few large ones. This allows the simple application of parallelism to the processing of queries in ADIM.

The choice of B-trees as the unique file structure throughout the data base management system enabled me, as designer, to avoid the unnecessary accumulation and manipulation of statistics, usually required for monitoring the efficiency of the system. The evaluation tactics described in chapters 5 and 6 make use of the properties of B-trees to estimate data flow. This obviously leads to good cost estimation of queries, updates and insertions in the data bases administered by ADIM.

The fulfilment of the requirements for expandability and flexibility demanded of ADIM, are demonstrated in

chapter 6. Applications such as the Examination Monitoring System are an illustration of this point.

Further work to be undertaken as well as some open problems emanate from the following list:

(a) Methods for using functional dependencies in the decomposition procedure.

(b) Use of security and integrity constraints in decomposition techniques.

(c) Cost criteria for optimizing relational expressions which include query algebra and decomposition operators.

(d) The use of several processors should be incorporated in the cost functions, by considering parallel processing. A distinction has to be made between one-site multiprocessor systems and distributed systems, because of different cost structures imposed by data pathways.

Finally, I feel that ADIM contributes an architecture and an implementation for a Desk-top Information Manager which is small and yet efficient. Furthermore, ADIM provides a flexible and expandable base for experimentation and development of new ideas in the areas of relational data base management systems, query languages, heterogeneous and homogeneous distributed data base systems and data base design.

APPENDICES

Note: The syntax of the languages used in the demonstrations might be slightly different from the syntax defined in Chapter 3. This is because an older version of ADIM was used in some of these demonstrations.

A. APPENDIX

ALFRED Demonstration

In this appendix, a short sequence of queries is presented. The queries involve relations created in APPENDIX B. The system's catalogues are examined often so a detailed picture of the processing of the queries can be observed. The queries are followed by a display of their evaluation.

```
% prolog
PROLOG Version NU7


?- ['alfred'].

alfred consulted.

yes
?- alfred(dept,query).
```

W E L C O M E

T O

A L F R E D

QUERIES :

```
display relation ?
display attribute ?

display staff ?
display addresses ?

join staff * addresses where
        snumber = staff_id
into t1 ?

display t1 ?

project t1:staff_id, name, salary, city
        into homes ?

display relation ?
display attribute ?


display homes !
```

ALFRED EXECUTION

RELATION: relation

relid	relsave	reltups	relwid	relatt	dvc	root	n
addresses	0	3	46	4	97	6	5
attribute	0	23	34	7	97	4	7
relation	0	4	28	8	97	1	8
staff	0	2	34	4	97	5	7

RELATION: attribute

aname	rid	format	asize	start	relati	keypos
staff_id	addresses	105	4	0	0	1
addr	addresses	115	20	4	1	0
city	addresses	115	10	24	2	0
phone	addresses	115	12	34	3	0
aname	attribute	115	12	0	0	0
rid	attribute	115	12	12	1	1
format	attribute	116	1	24	2	0
asize	attribute	104	2	26	3	0
start	attribute	104	2	28	4	0
relative	attribute	104	2	30	5	2
keypos	attribute	104	2	32	6	0
relid	relation	115	12	0	0	1
relsave	relation	105	4	12	1	0
reltups	relation	105	4	16	2	0
relwid	relation	104	2	20	3	0
relatts	relation	116	1	22	4	0
dvc	relation	116	1	23	5	0
root	relation	104	2	24	6	0
n	relation	116	1	26	7	0
snumber	staff	105	4	0	0	1
name	staff	115	20	4	1	0
room	staff	104	2	24	2	0
salary	staff	114	8	26	3	0

RELATION: staff

snumber	name	room	salary
87654	J. Jones	67	11000.000
123456	G. Smith	34	12345.500

RELATION: addresses

staff_id	addr	city	phone
87654	59 Richmond Rd	Bristol	24335
567436	22 Carnaby Rd.	London	234567
123456	34 Henry St.	Bristol	45678

RELATION: t1

snumber	name	room	salary	staff_id	addr
87654	J. Jones	67	11000.000	87654	59 Richmond R
123456	G. Smith	34	12345.500	123456	34 Henry St.

RELATION: relation

relid	relsave	reltups	relwid	relatt	dvc	root	n
addresses	0	3	46	4	97	6	5
attribute	0	35	34	7	97	4	7
homes	0	2	42	4	97	10	5
relation	0	4	28	8	97	1	8
staff	0	2	34	4	97	5	7
t1	0	2	80	8	97	8	3

RELATION: attribute

aname	rid	format	asize	start	relati	keypos
staff_id	addresses	105	4	0	0	1
addr	addresses	115	20	4	1	0
city	addresses	115	10	24	2	0
phone	addresses	115	12	34	3	0
aname	attribute	115	12	0	0	0
rid	attribute	115	12	12	1	1
format	attribute	116	1	24	2	0
asize	attribute	104	2	26	3	0
start	attribute	104	2	28	4	0
relative	attribute	104	2	30	5	2
keypos	attribute	104	2	32	6	0
staff_id	homes	105	4	0	0	1
name	homes	115	20	4	1	0
salary	homes	114	8	24	2	0
city	homes	115	10	32	3	0
relid	relation	115	12	0	0	1
relsave	relation	105	4	12	1	0
reltups	relation	105	4	16	2	0
relwid	relation	104	2	20	3	0
relatts	relation	116	1	22	4	0
dvc	relation	116	1	23	5	0
root	relation	104	2	24	6	0
n	relation	116	1	26	7	0
snumber	staff	105	4	0	0	1
name	staff	115	20	4	1	0
room	staff	104	2	24	2	0
salary	staff	114	8	26	3	0
snumber	t1	105	4	0	0	1
name	t1	115	20	4	1	0
room	t1	104	2	24	2	0
salary	t1	114	8	26	3	0
staff_id	t1	105	4	34	4	2
addr	t1	115	20	38	5	0
city	t1	115	10	58	6	0
phone	t1	115	12	68	7	0

RELATION: homes

staff_id	name	salary	city
87654	J. Jones	11000.000	Bristol
123456	G. Smith	12345.500	Bristol

LOCAL STACK 39
GLOBAL STACK 1887
FREE AREA 18884
TIME 279

yes
?-

B. APPENDIX

Utilities to the Data Base Administrator

A Demonstration

This appendix demonstrates some of the most important utilities available to the data base administrator. The creation of device 'a' is followed by the creation of the data base 'dept'. The effects of these actions in the file 'alldb' are shown. The invocation of ADIM demonstrates the facilities to create relations and to input data to relations. Also in this demonstration, the means to examine a relation and to manipulate the keys of such a relation are shown.

```
  cat ../alldb
data bases
devices
% mkdev
makedev: Usage
        makedev namedev size(in blocks)
% mkdev data 40
szdev = 40      mapsz = 1
mkdev -- in data 0 read and 0 write bad blocks found
*** device built and in good shape for use ***
% cat ../alldb
data bases
devices
        a       data    1       40
% dbmk
dbmk -- Usage:
        dbmk dbname device
% dbmk dept a
% cat ../alldb
data bases
        a       dept    1       4
devices
        a       data    1       40
%
```

```
% adim dept

                    A D I M

          A  Desk-top  Information  Manager

                Version 1.0


_display relation


RELATION:  relation

¦relid       ¦relsave   ¦reltups   ¦relwid¦relatt¦dvc    ¦root  ¦n      ¦
¦-------------------------------------------------------------------------¦
¦attribute   ¦         0¦        15¦     34¦      7¦     97¦      4¦      7¦
¦relation    ¦         0¦         2¦     28¦      8¦     97¦      1¦      8¦
¦-------------------------------------------------------------------------¦


_garbage

eh ?

_

eh ?

_
```

```
eh ?

_display attribute


RELATION:  attribute

!aname       !rid         !format!asize !start !relati!keypos!
!-------------------------------------------------------------!
!aname       !attribute   !   115!    12!     0!     0!     0!
!rid         !attribute   !   115!    12!    12!     1!     1!
!format      !attribute   !   116!     1!    24!     2!     0!
!asize       !attribute   !   104!     2!    26!     3!     0!
!start       !attribute   !   104!     2!    28!     4!     0!
!relative    !attribute   !   104!     2!    30!     5!     2!
!keypos      !attribute   !   104!     2!    32!     6!     0!
!relid       !relation    !   115!    12!     0!     0!     1!
!relsave     !relation    !   105!     4!    12!     1!     0!
!reltups     !relation    !   105!     4!    16!     2!     0!
!relwid      !relation    !   104!     2!    20!     3!     0!
!relatts     !relation    !   116!     1!    22!     4!     0!
!dvc         !relation    !   116!     1!    23!     5!     0!
!root        !relation    !   104!     2!    24!     6!     0!
!n           !relation    !   116!     1!    26!     7!     0!
!-------------------------------------------------------------!


_off
bye.

%
```

```
% adim dept

                        A D I M

                A  Desk-top  Information  Manager

                        Version 1.0


_create staff a



RELATION: staff

        Enter name and format for each attribute
        (CANCELATION:- Type: 0 after name-prom.)

name:  snumber

format:  i

more ? (y-n)  y

name:  name

format:  s20

more ? (y-n)  y

name:  room

format:  h

more ? (y-n)  y

name:  salary

format:  r

more ? (y-n)  n
```

```
        Is there a primary key  ? (y-n)  y


        Is the key compounded ? (y-n)  y


        Enter attribute names in decreasing order of importance.
        Type:
                2 - for HELP,
                1 - to FINISH,and
                0 - for CANCELATION.

        after the name-prom.

name:  2


        attributes are:

                snumber
                name
                room
                salary


name:  snumber

name:  1


        WARNING - single key !!


        confirm ? (y-n)y
```

```
create addresses a


RELATION: addresses

        Enter name and format for each attribute
        (CANCELATION:- Type: 0 after name-prom.)

name:  staff_id,

format:  i

more ? (y-n)  y

name:  addr

format:  s20

more ? (y-n)  y

name:  city

format:  s10

more ? (y-n)  y

name:  phone

format:  s12

more ? (y-n)  n
```

Is there a primary key ? (y-n) y

Is the key compounded ? (y-n) n

name: staff_id

_display relation

RELATION: relation

relid	relsave	reltups	relwid	relatt	dvc	root	n
addresses	0	0	46	4	97	-1	5
attribute	0	23	34	7	97	4	7
relation	0	4	28	8	97	1	8
staff	0	0	34	4	97	-1	7

```
display attribute
```

RELATION: attribute

aname	rid	format	asize	start	relati	keypos
staff_id	addresses	105	4	0	0	1
addr	addresses	115	20	4	1	0
city	addresses	115	10	24	2	0
phone	addresses	115	12	34	3	0
aname	attribute	115	12	0	0	0
rid	attribute	115	12	12	1	1
format	attribute	116	1	24	2	0
asize	attribute	104	2	26	3	0
start	attribute	104	2	28	4	0
relative	attribute	104	2	30	5	2
keypos	attribute	104	2	32	6	0
relid	relation	115	12	0	0	1
relsave	relation	105	4	12	1	0
reltups	relation	105	4	16	2	0
relwid	relation	104	2	20	3	0
relatts	relation	116	1	22	4	0
dvc	relation	116	1	23	5	0
root	relation	104	2	24	6	0
n	relation	116	1	26	7	0
snumber	staff	105	4	0	0	1
name	staff	115	20	4	1	0
room	staff	104	2	24	2	0
salary	staff	114	8	26	3	0

```
 display addresses


RELATION:  addresses

¦staff_id  ¦addr                ¦city      ¦phone       ¦
¦-----------------------------------------------------¦
¦-----------------------------------------------------¦


_display staff


RELATION:  staff

¦snumber   ¦name                ¦room  ¦salary        ¦
¦-------------------------------------------------¦
¦-------------------------------------------------¦


-
```

```
 append staff


 RELATION:  staff

Enter value for each attribute.

snumber       (i):  123456
name          (s):  G. Smith
room          (h):  34
salary        (r):  12345.5


_append staff



 RELATION:  staff

Enter value for each attribute.

snumber       (i):  87654
name          (s):  J. Jones
room          (h):  67xcd
room          (h):  67
salary        (r):  11000


_display staff



RELATION:  staff

¦snumber   ¦name                ¦room  ¦salary        ¦
¦-------------------------------------------------------¦
¦     87654¦J. Jones            ¦    67¦     11000.000¦
¦    123456¦G. Smith            ¦    34¦     12345.500¦
¦-------------------------------------------------------¦


_
```

```
append addresses



 RELATION:  addresses

Enter value for each attribute.

staff_id      (i):  123456
addr          (s):  34 Henry St.
city          (s):  Bristol
phone         (s):  45678


_append addresses



 RELATION:  addresses

Enter value for each attribute.

staff_id      (i):  567436
addr          (s):  22 Carnaby Rd.
city          (s):  London
phone         (s):  234567


_append addresses



 RELATION:  addresses

Enter value for each attribute.

staff_id      (i):  87654
addr          (s):  59 Richmond Rd
city          (s):  Bristol
phone         (s):  24335
```

```
display addresses


RELATION:  addresses

|staff_id  |addr                |city      |phone       |
|-----------------------------------------------------|
|     87654|59 Richmond Rd      |Bristol   |24335       |
|    567436|22 Carnaby Rd.      |London    |234567      |
|    123456|34 Henry St.        |Bristol   |45678       |
|-----------------------------------------------------|

_display staff


RELATION:  staff

|snumber   |name                |room  |salary        |
|-----------------------------------------------------|
|     87654|J. Jones            |    67|     11000.000|
|    123456|G. Smith            |    34|     12345.500|
|-----------------------------------------------------|


_
```

```
display relation


RELATION:  relation

!relid       !relsave   !reltups   !relwid!relatt!dvc    !root  !n     !
!-------------------------------------------------------------------------!
!addresses   !        0!         3!     46!     4!     97!     6!     5!
!attribute   !        0!        23!     34!     7!     97!     4!     7!
!relation    !        0!         4!     28!     8!     97!     1!     8!
!staff       !        0!         2!     34!     4!     97!     5!     7!
!-------------------------------------------------------------------------!
```

```
display attribute


RELATION:  attribute

!aname       !rid         !format!asize !start !relati!keypos!
!-------------------------------------------------------------!
!staff_id    !addresses   !   105!     4!     0!     0!     1!
!addr        !addresses   !   115!    20!     4!     1!     0!
!city        !addresses   !   115!    10!    24!     2!     0!
!phone       !addresses   !   115!    12!    34!     3!     0!
!aname       !attribute   !   115!    12!     0!     0!     0!
!rid         !attribute   !   115!    12!    12!     1!     1!
!format      !attribute   !   116!     1!    24!     2!     0!
!asize       !attribute   !   104!     2!    26!     3!     0!
!start       !attribute   !   104!     2!    28!     4!     0!
!relative    !attribute   !   104!     2!    30!     5!     2!
!keypos      !attribute   !   104!     2!    32!     6!     0!
!relid       !relation    !   115!    12!     0!     0!     1!
!relsave     !relation    !   105!     4!    12!     1!     0!
!reltups     !relation    !   105!     4!    16!     2!     0!
!relwid      !relation    !   104!     2!    20!     3!     0!
!relatts     !relation    !   116!     1!    22!     4!     0!
!dvc         !relation    !   116!     1!    23!     5!     0!
!root        !relation    !   104!     2!    24!     6!     0!
!n           !relation    !   116!     1!    26!     7!     0!
!snumber     !staff       !   105!     4!     0!     0!     1!
!name        !staff       !   115!    20!     4!     1!     0!
!room        !staff       !   104!     2!    24!     2!     0!
!salary      !staff       !   114!     8!    26!     3!     0!
!-------------------------------------------------------------!


_off
bye.

%
```

C. APPENDIX

FML Demonstration

A set of queries in FML is presented here. These queries are equivalent to the queries in the ALFRED demonstration [APPENDIX A]. An evaluation of the queries is also included.

```
#include "defs.h"
#include "global.h"

char *A14[] = { "snumber","=","staff_id",0 };
char *A1627[] = { "staff_id","name","salary","city",0 };
main() {
dbopen("dept");
display("relation",0);
display("attribute",0);
display("staff",0);
display("addresses",0);
join("t1","staff","addresses",A14,0);
display("t1",0);
project("homes","t1",A1627,0);
display("relation",0);
display("attribute",0);
display("homes",0);
dbclose();
}
%
```

RELATION: relation

relid	relsave	reltups	relwid	relatt	dvc	root	n
addresses	0	3	46	4	97	6	5
attribute	0	23	34	7	97	4	7
relation	0	4	28	8	97	1	8
staff	0	2	34	4	97	5	7

RELATION: attribute

aname	rid	format	asize	start	relati	keypos
staff_id	addresses	105	4	0	0	1
addr	addresses	115	20	4	1	0
city	addresses	115	10	24	2	0
phone	addresses	115	12	34	3	0
aname	attribute	115	12	0	0	0
rid	attribute	115	12	12	1	1
format	attribute	116	1	24	2	0
asize	attribute	104	2	26	3	0
start	attribute	104	2	28	4	0
relative	attribute	104	2	30	5	2
keypos	attribute	104	2	32	6	0
relid	relation	115	12	0	0	1
relsave	relation	105	4	12	1	0
reltups	relation	105	4	16	2	0
relwid	relation	104	2	20	3	0
relatts	relation	116	1	22	4	0
dvc	relation	116	1	23	5	0
root	relation	104	2	24	6	0
n	relation	116	1	26	7	0
snumber	staff	105	4	0	0	1
name	staff	115	20	4	1	0
room	staff	104	2	24	2	0
salary	staff	114	8	26	3	0

RELATION: staff

snumber	name	room	salary
87654	J. Jones	67	11000.000
123456	G. Smith	34	12345.500

RELATION: addresses

staff_id	addr	city	phone
87654	59 Richmond Rd	Bristol	24335
567436	22 Carnaby Rd.	London	234567
123456	34 Henry St.	Bristol	45678

RELATION: t1

snumber	name	room	salary	staff_id	addr
87654	J. Jones	67	11000.000	87654	59 Richm
123456	G. Smith	34	12345.500	123456	34 Henry

RELATION: relation

relid	relsave	reltups	relwid	relatt	dvc	root	n
addresses	0	3	46	4	97	6	5
attribute	0	35	34	7	97	4	7
homes	0	2	42	4	97	10	5
relation	0	4	28	8	97	1	8
staff	0	2	34	4	97	5	7
t1	0	2	80	8	97	8	3

RELATION: attribute

aname	rid	format	asize	start	relati	keypos
staff_id	addresses	105	4	0	0	1
addr	addresses	115	20	4	1	0
city	addresses	115	10	24	2	0
phone	addresses	115	12	34	3	0
aname	attribute	115	12	0	0	0
rid	attribute	115	12	12	1	1
format	attribute	116	1	24	2	0
asize	attribute	104	2	26	3	0
start	attribute	104	2	28	4	0
relative	attribute	104	2	30	5	2
keypos	attribute	104	2	32	6	0
staff_id	homes	105	4	0	0	1
name	homes	115	20	4	1	0
salary	homes	114	8	24	2	0
city	homes	115	10	32	3	0
relid	relation	115	12	0	0	1
relsave	relation	105	4	12	1	0
reltups	relation	105	4	16	2	0
relwid	relation	104	2	20	3	0
relatts	relation	116	1	22	4	0
dvc	relation	116	1	23	5	0
root	relation	104	2	24	6	0
n	relation	116	1	26	7	0
snumber	staff	105	4	0	0	1
name	staff	115	20	4	1	0
room	staff	104	2	24	2	0
salary	staff	114	8	26	3	0
snumber	t1	105	4	0	0	1
name	t1	115	20	4	1	0
room	t1	104	2	24	2	0
salary	t1	114	8	26	3	0
staff_id	t1	105	4	34	4	2
addr	t1	115	20	38	5	0
city	t1	115	10	58	6	0
phone	t1	115	12	68	7	0

RELATION: homes

staff_id	name	salary	city
87654	J. Jones	11000.000	Bristol
123456	G. Smith	12345.500	Bristol

%

D. APPENDIX

ALFRED-U to QUEL Translator

A Demonstration

Relations from the INGRES data base 'demo' are used in this demonstration. The relations and the file 'query' containing the ALFRED queries are shown. This is followed by the translation from ALFRED to QUEL and the evaluation of the queries.

```
  ingres demo
INGRES version 6.3/-1 login
Wed Jul  6 15:51:47 1983

COPYRIGHT
The Regents of the University of California
1977

This program material is the property of the
Regents of the University of California and
may not be reproduced or disclosed without
the prior written permission of the owner.

continue
* print item
* print supplier
* \g
```

```
Executing . . .

item relation

|number|name                |dept  |price |qoh   |suppli|
|-----------------------------------------------------|
|    26|Earrings            |    14|  1000|    20|   199|
|   118|Towels, Bath        |    26|   250|  1000|   213|
|    43|Maze                |    49|   325|   200|    89|
|   106|Clock Book          |    49|   198|   150|   125|
|    23|1 lb Box            |    10|   215|   100|    42|
|    52|Jacket              |    60|  3295|   300|    15|
|   165|Jean                |    65|   825|   500|    33|
|   258|Shirt               |    58|   650|  1200|    33|
|   120|Twin Sheet          |    26|   800|   750|   213|
|   301|Boy's Jean Suit     |    43|  1250|   500|    33|
|   121|Queen Sheet         |    26|  1375|   600|   213|
|   101|Slacks              |    63|  1600|   325|    15|
|   115|Gold Ring           |    14|  4995|    10|   199|
|    25|2 lb Box, Mix       |    10|   450|    75|    42|
|   119|Squeeze Ball        |    49|   250|   400|    89|
|    11|Wash Cloth          |     1|    75|   575|   213|
|    19|Bellbottoms         |    43|   450|   600|    33|
|    21|ABC Blocks          |     1|   198|   405|   125|
|   107|The 'Feel' Book     |    35|   225|   225|    89|
|   121|Ski Jumpsuit        |    65|  4350|   125|    15|
|-----------------------------------------------------|

supplier relation

|number|name          |city          |state |
|-------------------------------------------|
|   199|Koret         |Los Angeles   |Calif |
|   213|Cannon        |Atlanta       |Ga    |
|    33|Levi-Strauss  |San Francisco |Calif |
|    89|Fisher-Price  |Boston        |Mass  |
|   125|Playskool     |Dallas        |Tex   |
|    42|Whitman's     |Denver        |Colo  |
|    15|White Stag    |White Plains  |Neb   |
|-------------------------------------------|

continue
```

```
%



% cat query
/*

        IQL to QUEL example

*/
select item when `item.price >= 100´
        into T0 ?
project T0: `item=T0.name, T0.price, supno=T0.supplier´
        into T1 ?
project supplier: `supplier = supplier.name, supplier.number´
        into T2 ?
join T1 * T2 when
                `T1.supno = T2.number´
        into T3 ?
project T3: `T3.item, T3.supplier, T3.price´
        into highprice ?
display highprice ?
destroy T0, T1, T2, T3 ?
destroy highprice !
\p
\l
\g
%
```

```
% iql demo < query
/*

        IQL to QUEL example

*/
select item when `item.price >= 100´
        into T0 ?
project T0: `item=T0.name, T0.price, supno=T0.supplier´
        into T1 ?
project supplier: `supplier = supplier.name, supplier.number´
        into T2 ?
join T1 * T2 when
                `T1.supno = T2.number´
        into T3 ?
project T3: `T3.item, T3.supplier, T3.price´
        into highprice ?
display highprice ?
destroy T0, T1, T2, T3 ?
destroy highprice !
/*

        IQL to QUEL example

*/

range of item is item
retrieve into T0(item.all)
        where item.price >= 100

range of T0 is T0
retrieve into T1(item=T0.name, T0.price, supno=T0.supplier)

range of supplier is supplier
retrieve into T2(supplier = supplier.name, supplier.number)

range of T1 is T1
range of T2 is T2
retrieve into T3(T1.all, T2.all)
        where T1.supno = T2.number

range of T3 is T3
retrieve into highprice(T3.item, T3.supplier, T3.price)
print highprice
destroy T0, T1, T2, T3
destroy highprice
```

highprice relation

item	supplier	price
1 lb Box	Whitman's	215
2 lb Box, Mix	Whitman's	450
ABC Blocks	Playskool	198
Bellbottoms	Levi-Strauss	450
Boy's Jean Suit	Levi-Strauss	1250
Clock Book	Playskool	198
Earrings	Koret	1000
Gold Ring	Koret	4995
Jacket	White Stag	3295
Jean	Levi-Strauss	825
Maze	Fisher-Price	325
Queen Sheet	Cannon	1375
Shirt	Levi-Strauss	650
Ski Jumpsuit	White Stag	4350
Slacks	White Stag	1600
Squeeze Ball	Fisher-Price	250
The 'Feel' Book	Fisher-Price	225
Towels, Bath	Cannon	250
Twin Sheet	Cannon	800

% % % %

E. APPENDIX

ALFRED-U to QUEL Translator

Source Code

```
/*
        Translate IQL into QUEL
*/
/*
        SUGAR
*/
{define; ?;}
{define; |;}
{define; {continuetrap};\
        {type IQL query executed by INGRES ... \n\n\n}
/*
        COMMANDS
*/
/* UNION */
{define; union $1 + $2; \
range of $2 is $2 \
append to $1($2.all)}
/* PROJECT */
{define; project $r : $| into $t; \
range of $r is $r \
retrieve into $t($|)}
/* SELECT */
```

```
{define; select $r when $c into $t; \

range of $r is $r \

retrieve into $t($r.all) \
        where $c}

/* JOIN */

{define; join $1 * $2 when $c into $t; \

range of $1 is $1 \

range of $2 is $2 \

retrieve into $t($1.all, $2.all) \
        where $c}

/* DISPLAY */

{define; display; print}

/* OTHER COMMANDS ARE IDENTICAL */
```

A demonstration run of this translator is presented in APPENDIX

F. APPENDIX

Binary Cyclic Codes

Binary Cyclic Codes (BCC)

****

Definition 1.-

An (n,k) linear code C is called a cyclic code if it has the following property: If an n-tupla

$$v = (v_0, v_1, \ldots, v_{n-1})$$

is a code vector of C, the n-tupla

$$v^{(i)} = (v_{n-i}, v_{ni-i+1}, \ldots, v_{n-1}, v_0, v_1, \ldots, v_{n-i-1})$$

obtained by shifting v to the rigth cyclically i places, is also a code vector of C.

A relationship between the components of a code vector and the cofficients of a polynomial can be established, as follows:

$$v = (v_0, v_1, \ldots, v_{n-1}) \iff$$
$$v(X) = v_0 + v_1 X^1 + \ldots + v_{n-1} X^{n-1}$$

We shall call v(X) the code polynomial of v.

It can be shown easily that $v^{(i)}(X)$ is the remainder resulting from dividing $X^i v(X)$ by X^n, i.e.

$$X^i v(X) = q(X)(X^n + 1) + v^{(i)}(X)$$

It is clear that $v^{(i)}(X) = X^i v(X)$ if the degree of $X^i v(X)$ is n-1 or less.

Theorem 1.-

In an (n,k) cyclic code, there exists one and only one code polynomial g(X) of degree n-k

$$g(X) = 1 + g_1 X + g_2 X^2 + \ldots + g_{n-k-1} X^{n-k-1} + X^{n-k}$$

Every code polynomial v(X) is a multiple of g(X) and every polynomial of degree n-1 or less which is a multiple of g(X) must be a code polynomial.

It follows from Theorem 1 that for all v(X) in an (n,k) cyclic code

$$v(X) = m(X)g(X)$$
$$= (m_0 + m_1 X + m_2 X^2 + \ldots + m_{k-1} X^{k-1}) g(X)$$

If the coefficients of m(X), $(m_0,m_1,\ldots,m_{k-1})$ are the k information digits to be encoded, then v(X) would be the corresponding code polynomial. Thus,the encoding of a message m(X) is equivalent to multiplying the message m(X) by g(X). The polynomial g(X) is called the generator polynomial of the cyclic code. The degree n-k of g(X) is equal to the number of parity check digits of the code.

Theorem 2.-

The generator polynomial g(X) of an (n,k) cyclic code is a factor of X^n+1,i.e.

$$X^n+1 = g(X)h(X)$$

Theorem 3.-

If g(X) is a polynomial of degree n-k and is a factor of X^n+1, then g(X) generates an (n,k) yclic code.

Given the generator polynomial g(X) of an (n,k) cyclic code, the code can be put into systematic form. That is, the first k digits of each code word are the unaltered information digits; the last n-k digits are parity check digits.

Suposse that the message of k digits to be encoded is

$$m = (m_0, m_1, \ldots, m_{k-1})$$

The corresponding message polynomial is

$$m(X) = m_0 + m_1 X^1 + \ldots + m_{k-1} X^{k-1}$$

Multiplying m(X) by X^{n-k}, we obtain

$$X^{n-k} m(X) = q(X)g(X) + r(X) \quad (*)$$

where q(X) and r(X) are the quotient and remainder respectively.

Since the degree of g(X) is n-k the degree of r(X) must be n-k-1 or less,

$$r(X) = r_0, r_1 X^1 + \ldots + r_{n-k-1} X^{n-k-1}$$

Rearranging the equation marked by (*) above, we obtain

$$r(X) + X^{n-k} m(X) = q(X)g(X)$$

Thus by Theorem 1, $r(X) + X^{n-k} m(X)$ is a code polynomial generated by g(X). Writing out $r(X) + X^{n-k}$, we have

$$r(X) + X^{n-k} m(X) =$$

$$r_0+r_1X^1+\ldots+r_{n-k-1}X^{n-k-1}$$
$$+m_0X^{n-k}+m_1X^{n-k+1}+\ldots+m_{k-1}X^{n-1}$$

which corresponds to the code word

$$(r_0,r_1,\ldots,r_{n-k-1},m_0,m_1,\ldots,m_{k-1})$$

parity check----------message

G. APPENDIX

Cyclic Codes Algorithms

A Sample

```
/*
 *
 *        This program simulates a hardware encoding
device.
 *        The logic is based in Binary Cyclic Codes.
 *        The hardware device simulated is a
shift-register.
 *
*/

¬define BYTE              8
¬define N                 7
¬define K                 4

char    M[K / BYTE + 1]          { Ø13 };
char    G[(N - K) / BYTE + 1]    { Ø5 };
char    REM[(N - K) / BYTE + 1] { Ø };

main() {
        printf("message is ");
        output(M,K);
        printf("code generator is 1");
        output(G,N-K);
        p_rem(M,G,REM,N,K);
        printf("parity check bits for message are ");
        output(REM,N-K);
}

p_rem(m,g,rem,n,k)
char m[],g[],rem[];
int n,k;
{
        int i,j,top;
        char input,c;

        top = (n-k)/BYTE;

        for(i=Ø; i < k ;i++) {
                input = bit(m,k);
                c = (rem[top]   input) & Ø1 ? ~Ø : Ø;

                for(j=Ø; j <= top ;j++)
                        rem[j] =  (c & g[j]);

                r_shift(rem,n-k);

                rem[Ø] =i (c ? Ø1 << ((n-k-1) % BYTE) :
Ø);
        }

}
```

```
bit(a,s)
char a[];
int s;
{
        char t;

        t = a[s/BYTE] & 1;
        r_shift(a,s);
        return(t);
}

r_shift(a,s)
char a[];
int s;
{
        int top,i;

        top = s / BYTE;

        for(i=top; i > 0 ; i--) {
                    a[i] =>> 1;
                    a[i] =| (a[i-1] & 01 ?
                      (01<<(BYTE - 1)):0);
        }
        a[0] =>> 1;

}

output(a,s)
char a[];
int s;
{
        int i,top;

        top = s / BYTE;

        bitp(a[0],s % BYTE);

        for(i=1; i <= top ;i++)
                 bitp(a[i],BYTE);

        putchar('0);
}

bitp(pattern,size)
char pattern,size;
{
        char tester;

        if( size <= 0 ) return;

        tester = 01 << (size -1);

        while(size--) {
```

```
                putchar( tester & pattern ? '1' : '0');
                tester =>> 1;
            }
        }
```

H. APPENDIX

ALFRED VC to K Translator

Source Code

```
/*
decomp.pro
*/


?- op( 7, xfy, @ ).
?- op( 9, yfx, :!: ).
?- op( 10, yfx, :*: ).
?- op( 8, xfy, ^ ).


/*
        Map expression E into the fully
        decomposed and optimized expression F
*/

map( E, F ) :-
        char( E, E1 ),  /* add characteristic */
        expl( E1, E2 ),  /* explode views */
        simp( E2, F ).  /* optimize */



/*
        expand views to basic relations expressions
*/

expl( E, E ) :-  /* catch basic but not relations */
        basic( E ), !.


expl( E, F ) :-
        view( E, E1 ),
        expl( E1, F ).

expl( E, F ) :-
        E =.. [Op, Lexp, Rexp],
        expl( Lexp, Xexp ),
        expl( Rexp, Yexp ),
        F =.. [Op, Xexp, Yexp].

/*
        add characteristic to relations
*/

char( E, F ) :-
        characteristic( E, C ),
        F =.. [ @, E, C ], !.

char( E, F ) :-
        E =.. [ Op, Lexp, Rexp ],
        char( Lexp, Xexp ),
        char( Rexp, Yexp ),
        F =.. [Op, Xexp, Yexp].

char( E, E ).



/*
        Relational Optimiser
```

```
*/



/*
        simplify rel expr.
*/

simp( E, E ) :-
        basic( E ), !.

simp( E, F ) :-
        E =.. [ Op, Lexp, Rexp ],
        simp( Lexp, Xexp ),
        simp( Rexp, Yexp ),
        s( Op, Xexp, Yexp, F ).


basic( X ) :-
        relation( X ), !.

basic( X ) :-
        /* list of conditions or attributes */
        is_list( X ), !.


/*
        simplification rules
*/

/*
        restriction rules
*/

/* normalize relational expressions */

/* distribute @ over :+: */
s( @, R1 :+: R2, X, Z1 :+: Z2 ) :-
        s( @, R1, X, Z1 ),
        s( @, R2, X, Z2 ).



/* push @ to right and ^ to left */
s( ^, X @ Y, Z, X ^ Z @ Y ).



/* normalize predicate if necessary, then
   optimize: srestr( ) */

s( @, X, Y, Z ) :-
        clauseform( X, X1 ),
        clauseform( Y, Y1 ),
        srestr( @, X1, Y1, Z ).

/* empty relation => empty relation */
srestr( @, [], _, [] ).

/* empty condition => relation */
srestr( @, X, [], X ).
```

```
/* false condition => empty relation */
srestr( @, X, [false], [] ).



/* true condition => relation */
srestr( @, X, [true], X ).

/* associative case */
srestr( @, X, Y, Z ) :-
        is_list( X ),
        is_list( Y ),
        union( X, Y, Z1 ),
        set( Z1, Z2 ),  /* eliminate duplicates */
        optclauses( Z2, Z ).  /* optimize clauses again */

srestr( @, R @ X, Y, Z ) :-
        is_list( X ),
        is_list( Y ),
        union( X, Y, Z1 ),
        set( Z1, Z2 ),  /* eliminate duplicates */
        optclauses( Z2, Z3 ),  /* optimize clauses again */
        srestr( @, R, Z3, Z).  /* once more */


/* catch all */
srestr( @, X, Y, X @ Y ).



/*
        project rules
*/

/* empty relation => empty relation */
s( ^, [], _, [] ).

/* empty list of atts => empty relation */
s( ^, _, [], [] ).

/* associative case */
s( ^, X, Y, Z ) :-
        is_list( X ),
        is_list( Y ),
        intersection( X, Y, Z1 ),
        set( Z1, Z ). /* eliminate duplicates */

/* catch all */
s( ^, X, Y, X ^ Y ).


/*
        union rules
*/

/* relation :+: empty relation => relation */
s( :+:, X, [], X ).
s( :+:, [], X, X ).

/* catch all */
s( :+:, X, Y, X :+: Y ).
```

```
/*
        join rules
*/

/* relation :*: empty relation => empty relation */


s( :*:, _, [], [] ).
s( :*:, [], _, [] ).

/* catch all */
s( :*:, X, Y, X :*: Y ).



/*
        miscellaneous
*/

is_list( [] ).
is_list( [ _ | _ ] ).




/*
        Sets
*/

/* empty */
empty( [] ).

/* member */
member( X, [ X | _ ] ).
member( X, [ _ | Y ] ) :-
        member( X, Y ).

/* subset */
subset( [], _).
subset( X, X ).
subset( [ X | R ], Y ) :-
        member( X, Y ),
        !,
        subset( R, Y ).

proper( X, Y ) :-
        subset( X, Y ),
        not( subset( Y, X ) ).

equivalent( X, Y ) :-
        subset( X, Y ),
        subset( Y, X ).

/* intersection */
intersection( [], X, [] ).
intersection( [ X | R ], Y, [ X | Z ] ) :-
        member( X, Y ),
```

```
        !,
        intersection( R, Y, Z ).
intersection( [ X | R ], Y, Z ) :-
        intersection( R, Y, Z ).

/* union */
union( [], X, X ).
union( [ X | R ], Y, Z ) :-
        member( X, Y ),
        !,



        union( R, Y, Z ).
union( [ X | R ], Y, [ X | Z ] ) :-
        union( R, Y, Z ).


/* difference : relative complement */
difference( X, [], X ).
difference( [], _, [] ).
difference( [ X | XT ], Y, [ X | Z ] ) :-
        not member( X, Y ),
        !,
        difference( XT, Y, Z ).
difference( [ X | XT ], Y, Z ) :-
        member( X, Y ),
        !,
        difference( XT, Y, Z ).

/* disjoint */
disjoint( X, Y ) :-
        not( (member( Z, X ), member( Z, Y ) ) ).



/*
        Convinience - General
*/

/* delete all ocurrances of X from list L */
delete( _, [], [] ).
delete( X, [ X | L ], M ) :-
        !,
        delete( X, L, M ).
delete( X, [ Y | L1 ], [ Y | L2 ] ) :-
        delete( X, L1, L2 ).


/* make a set S from a list L:
   remove duplicates */
set( [], [] ).
set( [ X | S1 ], [ X | S ] ) :-
        not member( X, S1 ), !, set( S1, S ).
set( [ X | S1 ], S ) :-
        member( X, S1 ), !, set( S1, S ).



/* define operators */
```

```
?- op( 600, fx, ~ ).
?- op( 900, xfy, or ).
?- op( 900, xfy, and ).


/*
        normalize predicate calculus expression
*/


/*
clauseform( X, Y ) :-
        transform X expression to clause form Y



           if not already done. */
clauseform( X, Y ) :-
        not clform( X ),
        pcnorm( X, Y ).
clauseform( X, X ).

clform( [] ).
clform( [ true ] ).
clform( [ false ] ).
clform( [ cl(_,_) | _ ] ).


/*
        pcnorm ---
                normalises a predicate calculus expression
*/

pcnorm( [ Expression ], Clauses ) :-
        negin( Expression, X1 ),
        conjn( X1, X2 ),
        clausify( X2, X3, [] ),
        optclauses( X3, Clauses ).  /* optimize clauses */


/* move negation inwards */
negin((~P), P1) :-
        !, neg( P, P1 ).
negin((P and Q), (P1 and Q1)) :-
        !, negin( P, P1 ),
        negin( Q, Q1 ).
negin((P or Q), (P1 or Q1)) :-
        !, negin( P, P1 ),
        negin( Q, Q1 ).
negin( P, P ).

neg((~P), P1) :-
        !, negin( P, P1 ).
neg((P and Q), (P1 or Q1)) :-
        !, neg(P, P1),
        neg(Q, Q1).
neg((P or Q), (P1 and Q1)) :-
        !, neg(P, P1),
        neg(Q, Q1).
neg( P, (~P) ).


/* distribute and over or */
```

```
conjn((P or Q), R) :- !,
        conjn(P, P1),
        conjn(Q, Q1),
        conjnl((P1 or Q1), R).
conjn((P and Q), (P1 and Q1)) :- !,
        conjn(P, P1), conjn(Q, Q1).
conjn( P, P ).

conjnl(((P and Q) or R), (P1 and Q1)) :-
        !, conjn((P or Q), P1),
        conjn((Q or R), Q1).
conjnl((P or (Q and R)), (P1 and Q1)) :- !,
        conjn((P or Q), P1),
        conjn((P or R), Q1).
conjnl( P, P ).




/* into clauses */
clausify((P and Q), C1, C2) :-
        !, clausify(P, C1, C3),
        clausify(Q, C3, C2).
clausify(P, [ cl( A, B ) | Cs ], Cs) :-
        inclause( P, A, [], B, [] ), !.
clausify( _, C, C).

inclause((P or Q), A, A1, B, B1) :- !,
        inclause( P, A2, A1, B2, B1),
        inclause( Q, A, A2, B, B2).
inclause((~P), A, A, B1, B) :- !,
        notin( P, A), putin( P, B, B1).
inclause( P, A1, A, B, B) :-
        notin( P, B), putin( P, A, A1).

notin( X, [ X | _ ] ) :- !, fail.
notin( X, [ _ | L ] ) :- !,
        notin( X, L).
notin( X, [] ).

putin( X, [], [ X ] ):- !.
putin( X, [ X | L], L ) :- !.
putin( X, [ Y | L], [ Y | L1] ) :-
        putin( X, L, L1).



/*
        optclauses --
                optimize an expression in clause form.
                It finds contradiction in clauses, etc.
*/

optclauses( Clauses, OptClauses ) :-
        rmcontrary( Clauses, ShortCls ),
        optcls( ShortCls, OptClauses ).


/* rmcontrary - search for contradiction and if one
   is found, the contradictory clauses are maped
   into 'false' */
```

```
rmcontrary( [], [] ).
rmcontrary( [ cl( A, B ) | Cls ], [ false ] ) :-
        contrary( cl( A, B ), Cls ).  /* test for contrad. */
rmcontrary( [ cl( A, B ) | Cls1 ], [ cl( A, B ) | Cls2 ] ) :-
        rmcontrary( Cls1, Cls2 ).

/* contrary - test for contradiction */
contrary( cl( A, B ), [ cl( B1, A1 ) | _ ] ) :-
        equivalent( A, A1 ),
        equivalent( B, B1 ).
contrary( cl( A, B ), [ _ | Cls ] ) :-
        contrary( cl( A, B ), Cls ).

/* optcls -  false & X => false */
optcls( X, [ false ] ) :-
        member( false, X ).
/* optcls - true & X => X */
optcls( X, [ true ] ) :-



        member( true, X ).
optcls( X, X ).  /* catch all */




/*
        data base
*/

view( pupils, student ^ name ).
view( highpaid,
                (employee @ [salary >= 100]) ^ [name, salary] ).
view( manualworker,
     employee@[~(dept = 7) or (dept = 9 and section = 32)] ).
view( all_staff,
                dept1_staff :+: dept2_staff ).
view( dept1_staff,
                sectionA_staff :+: sectionB_staff ).



relation( employee ).
relation( sectionA_staff ).
relation( sectionB_staff ).
relation( dept2_staff ).
relation( student ).
relation( staff ).

characteristic( sectionA_staff, [section = 'A'] ).
characteristic( sectionB_staff,[~(section = 'A')] ).
```

REFERENCES

ADIBA

Adiba, M., Caleca, Y.J., and Euzet, C., "A distributed data base system using logical relational machines", Proc. 4th Conf. on Very Large Databases, Berlin, 1978.

ASHTON

ASHTON-TATE. "DBaseII Assembly-Language, Relational Database Management System". Reference Manual. Ashton-Tate. California. 1981.

BABB79

E. Babb. "Implementing a relational database by means of specialized hardware". ACM Transac. Database Syst. 4, 1. March 79. pp. 1-29.

BAYER

Bayer, R., and McCreight, E., "Organization and Maintenance of large ordered indexes". Acta Informatica 1,3 (1972), pp. 173-189.

BOCCA

J. Bocca. "On the Design of Personal Data Base Systems". IUCC-81 Colloquium, September 1981.

BOCCA82

J. Bocca. "Control of Distributed System - CDS". Internal Memo. Dept. of Computer Science, Bristol University. December 1982.

BURKHARD

Burkhard, W.A., "Hashing and Trie Algorithms for Partial Match Retrieval", ACM TODS 1, 2, June 1976, pp. 175-187.

CHAM

Champine, G.A., "Six Approaches to Distributed Databases", Datamation 23, 5, May 1977, pp. 69-72.

CLOMEL

W. Clocksin and C. Mellish. "Programming in Prolog", Springer-Verlag, 1981.

CODD70

E.F. Codd. "A Relational Model of Data for Large Shared Data Banks". CACM, 13, 1970. pp. 377-387.

CODD72

E.F. Codd. "Relational Completeness of Data Base Languages". IBM Research Report RJ909. New York, March 72.

COMMER79

D. Comer. "The Ubiquitous B-tree", Computing Surveys, Vol. 11, No.2, June 79.

CPM

CP/M Operating System, Digital Research Corp.

DATE

Date, C.J., "An Introduction to Data Base Systems", 3rd ed., Addisson-Wesley, 1981.

DEPPE

Deppe, M.E., and Fry, J.P., "Distributed Data Bases: A summary of research", Computer Networks 1, 2, Sept. 1976, pp. 130-138.

FAGIN

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H.R., "Extendible Hashing - A Fast Access Method for Dynamic Files", ACM TODS 4, 3 (1979), pp. 315-344.

FREDKIN

Fredkin, E.H., "Trie Memory", Communications of the ACM, Vol. 3, No. 9, 1960, pp. 490-499.

GALLAIRE

Gallaire, H., and Minker, J., "Logic and Data Bases", Plenum Press, 1978.

GUDES

Gudes, E., and Tsur, S., "Experiments with B-Tree Reorganization", Proc. ACM SIGMOD 1980, pp. 200-205.

HELD75

D.G. Held. "Storage Structures for Relational Data Base Management Systems". Memo No. ERL-M533. Electronics Research Laboratory - University of California, Berkeley, August 75.

HELLER

Heller, J., and Osterer, L., "The Design and Model of the BNL Archive and Dissemination System", Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977, pp. 161-181.

HELSTO75

G. Held and M. Stonebraker. "Storage Structures and Access Methods in the Relational Data Base Management System INGRES". Memo No. ERL-M505. Electronics Research Laboratory - University of California, Berkeley, March 1975,.

HEVNER

A. Hevner. "The Optimization of Query Processing on Distributed Database Systems", Ph.D. Dissertation, Purdue University, 1979.

HEVNERY

Hevner, A.R. and Yao, S.B., "Query Processing in Distributed Database Systems", IEEE Transactions on Software Engineering, Vol. SE-5, 3, May 1979, pp. 177-187.

HONEYW

HONEYWELL. LEVEL 68 (Software). Reference Manuals: LINUS and MRDS.

HOPCROFT

Hopcroft, J., Aho, A., and Ullman, J., "Data Structures and Algorithms", Addisson-Wesley, 1983.

HOSA

Horowitz and Sahni, "Fundamentals of Computer Algorithms", Computer Software Engineering Series, PITMAN, 1978.

HS75

G. Held and M. Stonebraker. "B-trees re-examined", Memo No. ERL-M528. Electronic Research Laboratory, University of California, Berkeley, July 75.

HSW75

G. Held, M. Stonebraker and E. Wong. "INGRES - A Relational Data Base Management System". AFIPS - Conference Proceedings, Vol. 44, USA, 1975.

HUTT78

A. Hutt, Relational Data Base Management System. Willey. Dec. 1979.

IBM66

IBM Corp. "OS ISAM Logic". IBM, White Plains, N.Y., GY28-6618. 1966.

JACOBSON

Jacobson, B., "DataEase vs. Condor and dBase II", BYTE, October 1984, pp. 289-302.

JARKE

Jarke, M., Clifford, J., and Vassiliou, Y., "An optimizing Prolog Front-End to a Relational Query System", Proc. ACM SIGMOD 84, pp. 296-306.

JARVAS

Jarbe, M., Vassiliou, Y., "Coupling expert systems with database management system", in Reitman, W. (ed.), Artificial Intelligence Applications for Business, Ablex, Norwood, NJ, 1984, pp. 65-85.

JOHNSON

S.C. JOHNSON. "YACC - Yet Another Compiler-Compiler". Bell Telephone Laboratory. Murray Hill, N.J.

KNUTH

Knuth. "The Art of Computer Programming", Vol.3, Addisson-Wesley, 1973.

KOWALSKI

Kowalski, R., "Logic for Problem Solving", Nilsson, J. (ed.), Artificial Intelligence Series 7, North Holland, 1979, pp. 37-42.

LANG78

G. Langdon, Jr. "A note on associative processors for data management". ACM Trans. Database Syst. 3, 2. June 78.

LARSON

Larson, P.A., "Dynamic hashing", BIT, Vol. 18, 1978, pp. 184-201.

LITWIN78

Litwin, W., "Virtual Hashing: a dynamically changing hashing", Proc. 4th Int. Conf. on Very Large Data Bases, Berlin, 1978, pp. 517-523.

LITWIN81

Litwin, W., "Trie Hashing", ACM-SIGMOD 81.

MISTRESS

Mistress User Manual.

NAKAMURA

Nakamura, T., and Mizoguchi, T., "An Analysis of the Sotrage Utilization Factor in Block Split Data Structuring Schemes", Proc. 4th Int. Conf. on Very Large Databases, Berlin, 1978.

NICOLA

Nicolas, J.M., and Yagdanian, K., "Integrity checking in Deductive Data Bases", in Logic and Data Bases, Gallaire, H., and Minker, J. (ed.), Plenum Press, New York, 1978, pp. 325-344.

OSS

E. Ozkarahan, S. Schuster, and K. Sevcik. "Performance evaluation of a relational associative processor". ACM Trans. Database Syst. 2, 2. June 77. pp. 175-195.

PALERMO

R. PALERMO. "A Data Base Search Problem", Fourth International Symposium on Computer and Information Science (COINS IV), Miami Beach, Florida, 1972.

PECHERER

R. Pecherer. "Efficient Retrieval in Relational Data Base Systems", Ph.D. Dissertation, University of California, Berkeley, 1975.

PROTEUS

PROTEUS working papers. Coordination of the PROTEUS project: P. Stocker, Computing Centre, University of East Anglia. Norwich - UK. 1981-1983.

QUITZOW

Quitzow, K.H., and Klopprogge, M., "Space Utilization and Access Path Length in B-Trees", Inf. Systems, Vol. 5, pp. 7-16.

RJLK78

D.M. Ritchie, S.C. Johnson, M.E. Lesk, B.W. Kernighan. "The C programming language". The Bell System Technical Journal. July-August 78.

ROSENBERG

Rosenberg, A.L., and Snyder, L., "Time and Space Optimality in B-Trees", ACM TODS 6, 1 March 1981, pp. 174-183.

RT74

E. Ritchie and K. Thompson. "The UNIX Time-Sharing System". CACM, 17, 1974. pp. 365-375.

SAGIV

Y. Sagiv. "Optimization of Queries in Relational Databses", Ph.D. thesis, Department of Electrical Engineering and Computer Science, Princeton University, October 1978.

SKCWHC

Shi-Kuo Chang and Wu-Haung Chen. "A Methodology for Structured Database Decomposition". IEEE Transactions on Software Engineering. Vol.6. No.2. March 198Ø.

STOCKER

Stocker, P.M. et.al., Paper on PROTEUS. Proc. 3rd Nac. Conf. on Data Bases, UK., 1984.

STONENEUH

Stonebraker, M., and Neuhold, E., "A distributed data base version of INGRES", Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, May 1977, pp. 161-181.

STOROWE

M. Stonebraker and L.A. Rowe. "Observations on Data Manipulation Languages and their embedding in general purpose programming languages". Memo No. UCB/ERL M77/53. Electronics Research Laboratory - University of California, Berkeley. July 1977.

SWKH76

M. Stonebraker, E. Wong, P. Kreps and G. Held. "The design and Implementation of INGRES". Memo No. ERL-M577. Electronics Research Laboratory - University of California, Berkeley, January 1976.

TAMMINEN

Tamminen, M., "Order preserving extendible hashing and bucket tries", BIT 21, 1981, pp. 419-435.

ULLMAN

J. Ullman. "Principles of Data Base Systems". Computer Software Engineering Series, PITMAN, 1980.

VETMAD

M. Veter and R.N. Maddison. "Database Design Methodology". Prentice-Hall International. 1981.

WIEDERHOLD

Wiederhold, G., "Database Design", McGraw-Hill, 2nd edition, 1983.

WOODAL

J. Woodfill et.al. INGRES version 6.2. Reference Manual. Memo. No. UCB/ERL M79/43. Electronics Research Laboratory, University of California, Berkeley, May 1979.

WOYU

E. Wong and K. Youssefi. "Decomposition - A Strategy for Query Processing", ACM Trans. on Data-base Systems, Vol.1, No.3, pp. 223-241, 1976.

WUN

Wun, S.S., "On a High-Performance VLS] Solution to Database Problems", Ph.D. dissertation, Carnegie-Mellon Univ., 1981.

YAO

Yao, A., "On Random 2-3 Trees", Acta Informatica 9, 1978, pp. 159-17Ø.

YOUSSEFI

K. Youssefi. "Query Processing for a Relational Database System", Ph.D. Dissertation, University of California, Berkeley, 1978.

BIOGRAPHICAL NOTE

Jorge Bernardino Bocca was born in Santiago, Chile on 26th September, 1950. He graduated with a Mention in Mathematics from the Liceo de Aplicacion, Santiago, Chile, in 1968. He attended University of Chile (School of Economics), receiving a B.Sc. degree in Economics in May 1973. In 1973 he worked for the National Copper Corporation - CHILE (CODELCO) as Systems Analyst. From mid-1974 and part of 1975, Jorge was an Assistant Lecturer teaching "Information Systems" at the Universidad Nacional del Sur, Bahia Blanca, Argentina.

From October 1976 to July 1978, he attended St. Andrew's University, Scotland, receiving an M.Sc. in Computational Science for a thesis entitled "RAL - Relational Algebra Language". In August 1978 he joined the Institute of Hearing Research of the Medical Research Council, as a Systems Programmer. In the Spring of 1979, he started work at the Computer Studies Group of the University of Southampton. Here, under the supervisiion of Professor David W. Barron, he was a Research Assistant working on a project studying some efficiency problems of relational data base management systems. While working on this project, in October 1980, he registered as part-time Ph.D. student at Southampton University.

Early in 1982, he joined the Computer Science Department at Bristol University. As Research Assistant first and as Research Associate later, he was a member of a research team working in the area of heterogeneous distributed data base management systems.

Since June 1983, Jorge has been Lecturer in Computing Science at the University of Ulster, Coleraine, County Londonderry, Northern Ireland. He is a member of the British Computer Society.