The ALISP Relational Database[1]

Kurt Konolige

COINS Technical Report 77-9

October 1977

[1]This research was supported by the National Science Foundation under Grant DCR75-16098.

ACKNOWLEDGMENTS

I would like to thank Ed Riseman for the support he generously gave during the course of this project. I wish also to thank him, Dave Stemple, Vic Lesser, and Elliot Soloway for valuable criticisms on earlier drafts of this manuscript.

Abstract

This report documents a relational database embedded in the ALISP language /15/. The intent is to provide a relational database facility biased towards AI applications. Relations are implemented as LISP data structures, and LISP functions provide a complete set of query operations. The resulting system provides complete database support for large-scale AI projects, as well as other applications.

Because of the tight embedding of the database in LISP, relations can be viewed as an abstract data type for the LISP language. Such data typing simplifies data management and programming structures for representing information. The relational paradigm offers significant advantages over current AI databases in the areas of representation, complexity, and efficiency and ease of programming. In addition, a relational database can achieve the same capabilities as current AI languages: associative retrieval, generators, deduction of implicit information, and multiple simultaneous states (contexts) of the database.

Because it is implemented in LISP, the database is essentially portable.

Table of Contents

0. Introduction

The relational paradigm for databases has been thoroughly developed since its inception in 1970 /5, 6, 7, 8, 9/, mostly in the context of commercial applications. This technical report describes a database system, the ALISP Relational Database /alisp/, which uses the relational paradigm to provide a solution to some of the database problems which AI researchers have encountered. While the relational paradigm has been used before to attack some problems in this field /weyl,deduce,sweden/, the system described here is the first consistent, uniform application of a relational scheme to produce a database system oriented towards AI applications. Two interconnected design problems are solved: embedding a relational database in a general-purpose list-processing host language /1/, and providing database tools for the AI researcher.

The content of this technical report also has a dual aspect. On the one hand, theoretical issues relating to various parts of a database system are discussed where appropriate (where such discussion would be too lengthy or overly controversial, references to relevant literature are given). On the other hand, some aspects of this report are closer to a user manual, with examples of database functions and techniques. It is hoped that this hybrid approach will help in understanding the full power and elegance of the relational paradigm as employed in this database system.

Finally, work on this project is not yet complete. While most of the general design problems have been solved, not all of the implementation is complete. In addition, there are some relational database results which could be applied but for lack of time have not, e.g., declarative semantic constraints between relations. Where this occurs, the possibility of using various techniques in the future will be mentioned, along with the possible benefits to be gained from them.

I. Background and Motivation

Most of the discussion below concerns databases in recently developed AI languages; a good reference is /2/. Issues of control structure raised by some of these languages are largely superfluous to database issues, and will not be considered here.

As artificial intelligence projects have become more ambitious and complex in recent years, the need for database support for these projects has grown acute. A new generation of AI languages assumed some of this burden by providing structured or abstract data types. Typical among these were the triples of LEAP/SAIL; tuples of PLANNER/CONNIVER; and bags, tuples, and sets of QLISP /11, 17, 19, 20/.

The chief advantage of using structured data types lay in the flexible query facilities. The burden of cross-indexing and otherwise maintaining the database was taken from the programmer. A simple, declarative pattern gave access to the database. The database itself achieved the desirable goal of data independence, where the method of accessing the data was independent of the physical storage method. A user of these languages could program a database query once, confident that it would still work even if better storage access mechanisms were implemented at a later time. The systems designer was free to choose access mechanisms which optimized as large a set of queries as possible. The database could thus evolve and become more efficient over time without the problems associated with re-writing user programs.

The concept of data independence is thus central to a good database design, and has been recognized as such for a long time in the database field as well /10/. However, data independence and a flexible query regimen by themselves are no guarantee that a database will fulfill the needs for which it is designed. Current AI language databases suffer from deficiencies in several important areas of database support. While these deficiencies are not serious or even apparent for small-scale databases, they become acute when the size and complexity of the database grows. For this reason, the issues brought up here are taken from a large body of literature dealing with databases for commercial applications, as well as personal experience in setting up databases for several AI projects. It seems reasonable that an AI database could benefit from this experience.

Three fundamentally important areas of database design where AI language databases fall short are:

a) Representation. Languages such as LEAP or CONNIVER provide a single type of data structure (triples for LEAP, n-tuples for CONNIVER). It is often hard to express relevant information in a natural way using these data structures. Natural here means that the data structure does not support the semantics of the information represented. To be more specific: in order to represent a person's characteristics in the CONNIVER

database, we might use tuples of the form:

(name age height weight salary cigarette-type ...)

A particular instance of this type of tuple would be, e.g.,

(JOHN 23 59 145 8000 COOLS ...)

Now, we really don't want more than one tuple of this type hanging around for any particular person. If, simultaneously, we had:

(JOHN 23 59 145 8000 COOLS ...)
(JOHN 19 54 130 0 NIL ...)

then we would consider the database to be inconsistent. Many other types of inconsistency could also occur, e.g., a negative number for the age. The point is that by using tuples to represent personal information, problems of this sort can occur. What we would really like is a data structure in which these problems are impossible. Such a "typed" data structure would be a natural one for expressing the information (1). There is considerable current interest in typing data structures /12/; most current AI language databases lack this facility.(2)

b) Complexity. For large, intricate databases, the complexity of the stored information may itself become difficult for the programmer to handle. He has to keep track of the way many different types of information are represented, and must make sure that any changes he makes to the database maintain the semantic consistency of the data. Unfortunately, current AI databases lack even a rudimentary facility for describing the structure of the database (as opposed to its content). The system sees only a homogenous pool of tuples or some other data type, while the user knows much more about the way his data is grouped and constrained semantically, but has no way of communicating this to the system. Without some description of database structure, it is impossible for

---

(1)CONNIVER and some of databases offer a demon mechanism for checking of consistency problems such as this. This is not a very good solution, however, for a number of reasons. For arguments against demons in particular, see the section on demons. Note that even if demons were able to check for consistency, the system would still not be able to use typing as a means for efficient data storage.

(2)QLISP is a notable exception, but it only offers a variety of data structures, rather than a true typing facility /19/.

the user to declare semantic constraints and have the system maintain them.(3)

Problems of complexity can also arise in connection with information stored implicitly in the database. The issues involved here are more complex, and are covered in the section on Views as a Deductive Mechanism.

c) Efficiency. There are two separate issues here. The first concerns making the physical data structures which model the abstract data type (tuples, bags, or whatever) more efficient in terms of access time and storage requirements. While it is true that AI language databases have the property of data independence, they are prevented from using it to full advantage precisely because of the grouping problem mentioned above. There is no way for the programmer to group data which will be queried in the same manner; hence the system must try to make all queries to all data equally efficient, and can't optimize for the actual subset of queries used on given groups of data. For example, the personal information tuple used in (a) above is indexed by the CONNIVER database on all tuple elements. If the user is going to ask for these tuples only by person's name, then such extra indexing overhead is wasteful of storage and more time-consuming on insertion and deletion operations.(4)

A second issue concerns efficient utilization of secondary storage devices with slow access times and fast transfer rates. The most effective secondary storage scheme would be tied closely to the access mechanisms of the database, in order to minimize the number of secondary storage transfers and maximize the relevant information returned with each transfer. Again, since there is no way for the user to specify a logical grouping of data, the system doesn't know what data will be likely to be needed together, and can't partition secondary storage for reasonable efficiency /18/.

---

(3)Here again, some AI language databases use insertion and deletion demons to help the programmer. These are procedures which are invoked when a change to the database takes place. For several reasons, these demons are really not an adequate mechanism. For a more detailed criticism, see the section on demons.

(4)There is some work currently being done on optimizing data structure physical storage in SAIL by analyzing access functions during compilation /16/ . Such automation techniques should be even more effective if they can use grouping information supplied by the programmer.

There is some question as to whether the abovementioned problems are inherent in the approach to databases taken by current AI languages. What is certain is that major changes in the definitional facilities of these databases would have to be made in order to accomodate such properties as data typing, data grouping, or declarative semantic constraints. On the other hand, the relational paradigm, originally developed for large-scale databases in commercial applications, can provide a uniform, elegant solution to these problems. Specifically, the relational paradigm has significant advantages over current AI databases in all the previously mentioned areas:

a) Representation - the relational approach is founded on the theory of relations, or sets of n-tuples. Normal forms provide guidelines for embedding the semantics of the information to be represented directly in the relational structure. The relations form a natural semantic partitioning of the database.

b) Complexity - a high-level description of the database is present in the schema, which is a description of the time-invariant properties of relations. Semantic integrity constraints between relations can be imposed in a declarative manner. Information contained implicitly in the database can be declared in a uniform way using views.

c) Efficiency and ease of programming - the non-hierarchical structure of relations lends itself to many different methods of physical storage, with varying indexing characteristics, both in core and on mass storage devices. The physical storage method can be changed to make queries to the relation more efficient, without affecting the form of the query operations in programs.

These aspects of the relational paradigm have been recognized as important for AI databases, and there have been (at least) two experiments in providing support for the AI researcher /1, 21/ . However, the implementation described here is unique in two respects:

1) Relations are exploited as an abstract data type for the LISP language /12/. In this respect they continue the trend of AI languages in providing a uniform structure for representing information. There is no need for an interface between LISP and the relational database; relations are implemented as LISP data structures (lists, arrays, files) and database commands are LISP functions. Such a tight embedding should outperform an interface implementation in terms of efficiency, and has the advantage of simplicity, since relations are viewed as an extension of LISP's data

types, a la PLANNER's tuples.

2) This implementation is intended as a complete database system, and hence a total replacement for current AI language database facilities. Subsequent parts of this report show how relations, embedded in LISP, achieve the same capabilities as current AI languages -- associative retrieval, generators, deduction of implicit information, and multiple simultaneous states (contexts) of the database. Most of these ideas were drawn from current AI languages, but fit quite naturally into the relational scheme as well. A single relational database can thus provide two types of support needed by large AI projects: an efficient, easy-to-use large-scale information store; and a flexible model-building database environment.

While this implementation of the RDB was biased towards large AI projects, and specifically the VISIONS scene-interpretation system, it has other applications. It is especially good for symbolic manipulations on large databases, a requirement surfacing more often in the social sciences as computer use becomes more sophisticated. The ALISP RDB currently supports an analysis system for 19th century Russian censuses, and will shortly be used for an anthropological study of Rumanian villages /14/. The interpretive nature of LISP makes programming and debugging an easier task, while the relational database allows large amounts of information to be efficiently stored and accessed. Finally, a LISP RDB may be useful in a commercial environment, where there is a demand for custom-tailored stand-alone query systems. Such a system could be programmed and debugged at a relatively small cost, again because of the interactive, symbolic nature of LISP.

II. The Relational Data Structure

The relational data structure has been adequately described in a series of papers /5, 6, 8/. A readable account is in /10/. A brief description of the relational structure will be given here, and some properties which relate to its embedding in LISP will be examined in more detail. Finally, we reveal some common techniques for representing information in a LISP RDB.

II.1 Tuples and Relations

The basic unit of representation is the ordered n-tuple. In LISP, n-tuples become lists of items, where the order of the items is important:

(I1 I2 ... In)

Items can be any LISP data structures: atoms, other lists, arrays, etc. These lists should be familiar to users of PLANNER/CONNIVER, where they are called assertions.

N-tuples are all the structure there is in databases of assertions. In the relational paradigm, however, n-tuples of the same length are grouped together into named sets, called relations. All the n-tuples in a given relation have some common interpretation which the user associates with the relation (and which the relation name usually reflects). For example, a relation consisting of duples of the form:

(father child)

might be grouped into a relation called FATHER-OF. The interpretation of the relation is that it assigns fathers to their children.

Any n-tuple in the database resides in one and only one relation, since it is the relation which gives meaning to the n-tuple. It is thus impossible, for example, to insert a tuple into the database without putting it into a relation. All database operations on tuples will always be done with respect to a relation or set of relations. Since all n-tuples in a given relation are of the same length, we will refer to them simply as tuples.

Relations give a time-invariant structure to the database(5) which is independent of the time-varying content of the database, the tuples. The beauty of relations is that not only do they

---

(5)Or relatively time-invariant, since relations can be created and destroyed dynamically.

impose a syntactic order on the tuples, they also partition the database neatly in a semantic fashion, since each relation represents one particular type of information.

Relations are the key to both complexity reduction and efficiency: complexity reduction because it is possible to talk about the semantics of the database in terms of a few time-invariant relation definitions rather than individual tuples; and efficiency because relations cluster tuples which are semantically linked (6) . These properties of relations are discussed in following sections.

Tuples which are grouped into a relation can be viewed as an array of items:

```
(I11 I12 I13 ... I1n)
(I21 I22 I23 ... I2n)
 .
 .
 .
(Ik1 Ik2 Ik3 ... Ikn)
```

In practice, it is convenient to refer to an item without knowing its position in the tuple, and so for a given relation the positions are given unique names, called role names (again, semantics dictates that an appropriate name be chosen). A relation is then described by giving its name and the role names for items, e.g.,

```
FATHER-OF(FATHER,SON)
```

describes a relation of duples whose first item denotes a father, the second his son. The contents of the relation are conveniently given as a table whose columns are labelled by the role names. Each tuple of the relation is one row in the table:

FATHER-OF

FATHER	SON
JOE	PETE
JOE	GEORGE
JOHN	CHARLY
BRAD	JOE

Throughout this report a table will be used to mean the relation it represents, but the two terms will be differentiated where necessary. The same holds for row and tuple. Also, it will

---

(6)There are also semantic links between relations, and for these, auxiliary access paths in the physical data structure are often provided.

often be convenient to refer to the set of items in a particular position for all tuples in a relation; the term column will be used for this set of items.

II.2 Normalization

Normal forms are various syntactic and semantic rules imposed on the way information is stored within a relation. These rules serve as guidelines to the user in avoiding representation problems, and also enable the system to store data more efficiently. Detailed accounts of normal forms can be found in the references cited at the beginning of the chapter; here they are explained briefly, and their relevance to the LISP embedding is discussed.

II.2.1 First Normal Form (1NF)

Items of tuples can be arbitrary LISP S-expressions; however, it is convenient in most cases to limit items to structureless LISP atoms: literals, strings, and numbers. When this is the case, a relation is said to be in first normal form. First normal form is purely syntactic, and has the following advantages over unnormalized forms:

1) There is no complex substructure to the database. If structured items were used, a tuple could have, say, another tuple as one its items, and a network or hierarchical substructure would be created. Databases without a network or hierarchical structure are often simpler to understand (although this point depends on preference and is certainly arguable).

2) A relation in 1NF is a self-sufficient structure, and can't be side-affected by operations on other relations. In this respect, the 1NF is easier to

---

(7)For example, in a hierarchically structured database, deleting data from some point in the hierarchy can invalidate the structure above that point in the hierarchy, since data may have pointed at the deleted data. Similarly, the deleted data may have had pointers to data below it on the hierarchy, which data then becomes unreachable. In a 1NF relation, it is impossible to cause such a syntactically anomalous situation. Problems of semantic consistency, of course, still exist for both structures.

operate on than unnormalized structures.(7)

3) The relation can be stored more easily with a variety of in-core and secondary storage methods, since there are no pointers to worry about.

It may not be readily apparent, but it can be shown that relations in 1NF have the same representational power as unnormalized relations. We illustrate here one common technique for representing a set of values. For example, in the previously mentioned FATHER-OF relation, we might wish to give each father a set of children, rather than just one child. The unnormalized relation would look like:

FATHER-OF

FATHER	CHILDREN
MIKE	(TOM SUE JOE)
JOHN	(HARRY)
BOB	(JILL JOHN)

where the CHILDREN item can be a set of names. To convert to to 1NF is a simple matter; each child is put into one row, while a father may occur in several:

FATHER-OF-1NF

FATHER	CHILD
MIKE	TOM
MIKE	SUE
MIKE	JOE
JOHN	HARRY
BOB	JILL
BOB	JOHN

While it would seem that this method is less storage efficient than the first, both may be implemented by the same physical storage structures. Here we are only concerned about the way the data looks to the user, and how easily he can understand and manipulate it. The way the user looks at the data will be called the logical representation of the data.

In practice, it is sometimes awkward to stick to 1NF, and so the ALISP RDB will support arbitrary S-expresions as tuple items. However, using unnormalized relations will limit the user's options for physical storage of relations, and the effort should be made to put the representation into normal form.

II.2.2 Keys

A relation is a set, and thus there are never two copies of the same tuple in a given relation. If we return to the FATHER-OF example, it is easy to see that we need only a single unique copy of each row to retain the information in the relation; duplicate rows can be eliminated.

There are times when it is useful to make the uniqueness condition for tuples in a relation more strict: two tuples could be duplicates if they matched in only some of their items, but not all. To illustrate this, consider representing information about marital unions in a relation:

PARENTS(FATHER,MOTHER,YEAR)

FATHER and MOTHER are self-explanatory, and YEAR is the year of marriage. The uniqueness condition for tuples in this relation is that no two tuples should have the same values in both FATHER and MOTHER items. The value of YEAR shouldn't make any difference; if there is a tuple (JOE MARY 1936), then there shouldn't be one like (JOE MARY 1947) -- assuming, of course, that in our world there are no second marriages between the same people.

The items which are used to determine tuple uniqueness are called key items; the set of key item role names (FATHER,MOTHER) is called the key of the relation. The user can declare a key for a relation when he defines it.

Keys are a way of embedding semantics directly into the structure of a relation. In the example above, it is the peculiar nature of marriages which led to the choice of the key (FATHER,MOTHER). The representation will support remarriages, but not between the same people. Typical contents of the PARENTS relation with this key might be:

PARENTS

FATHER	MOTHER	YEAR
JOE	MARY	1934
JOE	SUE	1938
BRAD	MARY	1939

which says that JOE married MARY in 1934 and SUE in 1938, and BRAD married MARY in 1939. If JOE were to marry MARY again at some time later than 1934, there is no way to represent this remarriage without destroying the record of the first one.

On the other hand, suppose we wish to represent a world in which men may remarry, but women never do (not a bad model for some societies). Then we could enforce this condition by making (MOTHER) the key. This guarantees only one tuple for each different MOTHER item, hence, a woman can have only one husband.

It is obvious from these two cases that the choice of a key for a relation is motivated by the semantics of the information to be represented. Proper choice of relations and their keys can make it much easier to maintain semantically consistent data, since the database automatically upholds the key uniqueness condition for tuples in a relation.

When writing relation definitions in the remainder of this report, key item names will be underlined:

PARENTS(FATHER,MOTHER,YEAR)

The example relations in this section have all been in 1NF. However, the concept of a correctly identified key can be carried over to an unnormalized relation, although the key will in general be different from its 1NF counterpart. The two relations FATHER-OF (unnormalized) and FATHER-OF-1NF furnish an example here. Their respective keys would be:

FATHER-OF(FATHER,CHILDREN)

FATHER-OF-1NF(FATHER,CHILD)

In FATHER-OF, there is thus a single row for each father, which gives all the children of that parent. On the other hand, in FATHER-OF-1NF, a father appears in as many rows as he has children, and it is the father-child pair which is unique.

As is true in all the above examples, the key items of a relation are usually atomic. There are some cases where it may be desirable to use more general list structures as keys to a relation. It is expected that such cases will be rare, and reasonable alternative representations could be easily formulated. Thus the ALISP RDB will support only atomic keys, and the user should be aware of this restriction when defining relations.

II.2.3 Second and Third Normal Forms (2NF and 3NF)

2NF and 3NF for relations are rules which says essentially that only one type of information should be represented in a single relation. Consider, for example, adding to the PARENTS relation another type of information, say, the current age of each person. Then PARENTS might look like:

PARENTS

FATHER	MOTHER	AGEF	AGEM	YEAR
JOE	MARY	43	26	1934
JOE	SUE	43	33	1938
BRAD	MARY	29	26	1939

Since the current age of a person is not connected to that person's liaisons, putting it in the PARENTS relation creates an anomalous representation: there is, for example, no way to delete BRAD's liaison without losing his age, too. After a brief exposure to representing information via relations, it becomes easy to recognize such anomalous situations. The solution is to split the offending relation into two or more relations in 3NF. In the above example, from the AGEM and AGEF items we would form the new relation:

AGES

PERSON	AGE
JOE	43
MARY	26
SUE	33
BRAD	29

with key AGES(PERSON,AGE).

The key items in a 3NF relation are those items which are an essential part of the relation's information, e.g., the FATHER and MOTHER items completely define parents. The non-key items represent auxiliary information attached to each tuple, e.g., the year of the liaison. To be in 3NF, a relation must have its key correctly identified as the essential items of the relation.

II.3 Inter-table Relationships

A single type of information is represented in a single (normalized) relation. A database consists of any number of such relations. Additional information is often encoded implicitly by an interpretation of several relations which are considered to be semantically linked. The most common type of semantic linking occurs when two relations each have a column which is interpreted in the same way. A simple example of this type is given by the two relations:

PARENTS(FATHER,MOTHER,LID)

OFFSPRING(LID,CHILD)

PARENTS is the same as previously presented, except for the LID item. LID is an abbreviation for liaison identifier. It is an integer unique to each father-mother pair. The LID is used by other relations to refer to a particular row in the PARENTS relation. Thus, each row of the OFFSPRING relation assigns a child to a specific liaison via the LID. A family consisting of JOHN, his wife MARY, and children SUE and BRAD, would be represented as:

PARENTS

FATHER	MOTHER	LID
JOHN	MARY	3

OFFSPRING

LID	CHILD
3	SUE
3	BRAD

III. Relational Operations

There are three basic operations for changing the contents of relations: insertion, deletion, and modification. In addition, there are several retrieval operations, or queries, on relations.

All examples used in this section deal with two relations:

PARENTS(FATHER,MOTHER,LID)

OFFSPRING(LID,CHILD)

LID is an abbreviation for liaison identifier, a unique integer assigned to each father-mother pair. This integer can be present in OFFSPRING, linking the parents to their children. A family consisting of JOHN, his wife MARY, and children SUE and BRAD, would be represented as:

PARENTS

FATHER	MOTHER	LID
JOHN	MARY	3

OFFSPRING

LID	CHILD
3	SUE
3	BRAD

III.1 Insertion

Tuples are inserted into a relation one at a time. The action is as follows: the tuple is inserted into the relation, and any tuple with the same key is deleted.

Note that the insert operation preserves the keyness of the relation, by throwing away any tuple with the same values in the key columns.

In ALISP, the tuple is a list whose elements correspond to the item values. The order of elements in the list should be the same as the originally defined column order of the relation. For example, in the PARENTS relation, we might want to insert a tuple with:

FATHER = JOHN
MOTHER = MARY
LID = 3

Since the order of the columns is (FATHER,MOTHER,LID), the ALISP S-expression for the tuple to be inserted is:

(JOHN MARY 3)

The ALISP function is called INSERT. It takes two arguments: the first is the relation name, and is not evaluated; the second is the tuple, which is evaluated. To insert the above tuple into the PARENTS relation, use:

(INSERT PARENTS '(JOHN MARY 3))

The value of INSERT is the inserted tuple.

Two extensions of INSERT make it easier to use. If the tuple list is too short for the relation, it is right-filled with NIL's (if it is too long, an error is given). Secondly, an optional third argument to INSERT can specify the tuple by naming the columns in which the values are to go. Thus it is not necessary to remember column orderings, only their names. This third argument is not evaluated. All unnamed columns are given NIL values.

The following insertions all do the same thing:

```
(INSERT PARENTS '(JOHN MARY))
(INSERT PARENTS '(JOHN MARY NIL))
(INSERT PARENTS '(MARY JOHN) (MOTHER FATHER))
(INSERT PARENTS '(JOHN MARY NIL) (FATHER MOTHER LID))
```

III.2 Deletion

A subset of the tuples in a relation can be deleted with a single deletion operation. The method of specifying in general a subset of relation tuples is by a predicate, as is explained in the next section.

III.2.1 Predicates

Deletion and modification use predicates to subset or restrict a relation. The subsetting is accomplished by applying a predicate to each tuple of the relation, such that the tuple is included in the subset if the predicate is satisfied, and is excluded if not. Typically, the predicate uses the item names of the relation as free variables in forming a relational expression. For a specific tuple, the items of the tuple are substituted for the free variables, and the expression evaluated.

For example, in the PARENTS relation, a predicate might be:

FATHER = "JOHN" OR LID<3

which would select all tuples with JOHN in the first item or 1 or 2 in the third.

In embedding the RDB in ALISP, predicates become S-expressions which are evaluated for each tuple, and are satisfied if they return non-NIL, unsatisfied if they return NIL. The item names still play the role of free variables, so that when the S-expression is evaluated for a particular tuple, the variables take on the item values. The S-expression equivalent to the predicate above is:

(OR (EQ FATHER 'JOHN) (LESSP LID 3))

A particularly useful S-expression is the constant atom T, which selects all tuples of a relation.

The S-expression evaluation provides a particularly elegant solution to the problem of embedding predicates in ALISP. Because the S-expression is an ALISP procedure, it can be evaluated efficiently by the interpreter to find if it is satisfied by a given tuple. Also, any ALISP function can be used in the S-expression. The host ALISP language provides a reservoir of predicates, and a restricted sub-language of predicates for sole use by the RDB does not have to be defined.

The S-expression, can also be looked upon as a declarative predicate statement (it is one of the peculiarities of LISP that procedures and data share the same data form, namely, the S-expression). As such, it can be parsed by an optimizer to make the subsetting process more efficient. This is indeed done in deletion, modification, and querying, when an appropriate physical form of the relation is encountered (there is a later section explaining the simple optimizer currently in use).

III.2.2 Predicates vs. Patterns

Current AI languages generally use some sort of template or pattern to subset their database of tuples. This section shows that the two methods, predicate and pattern, are equivalent in subsetting power, although their form is different.

A pattern is an n-tuple in which the items may have "?" values. A tuple matches the pattern if the tuple items match the pattern items in all positions except where there is a "?". This simple type of pattern is equivalent to a conjunctive predicate,

for example:

```
(AND (EQ LID 3)                    (JOHN ? 3)
     (EQ FATHER 'JOHN') )
```

More generally, a pattern can bind items to variables and apply an arbitrary test to the bound variables. In this form, the pattern is obviously equivalent to the predicate, which also applies an arbitrary test to the pre-defined column name variables. For example:

```
(OR (LESSP LID 3)          (?X ? ?(Y
    (EQ FATHER 'JOHN))          (OR (LESSP $Y 3)
                                    (EQ $X 'JOHN')) ))
```

The choice of which to use, pattern or predicate, is an aesthetic one. In general, predicates will be more clumsy if there are few "don't care" values, and patterns more clumsy if the specification is more complex. Initially, only predicates will be allowed for RDB operations, but inclusion of patterns is possible, and will be done if there is a demand.

III.2.3 Simple Deletion

A subset of tuples of a relation can be deleted with a single deletion operation. The subset is defined by a predicate on the relation. Action is as follows:

Each tuple is checked against the predicate; if it satisfies the predicate, it is deleted. The deletion process continues until the relation is exhausted.

The ALISP function is called DELETE. It takes two arguments; the first is the relation name, and is not evaluated; the second is a predicate, and also is not evaluated:

```
(DELETE relname pred)
```

For example, to delete all rows of the PARENTS relation for which the father is JOHN and the liaison identifier is 2, use:

```
(DELETE PARENTS (AND (EQ FATHER 'JOHN) (EQ LID 2)))
```

The value of DELETE is the number of tuples deleted from the relation, as an SNUM.

III.2.4 Response Forms

The response form gives the user greater control over the deletion and modification operations. The response form is simply an additional argument to the operation. Each time the operation's predicate returns true (non-NIL), the response form is evaluated. The exact effect of the resonse form depends upon the operation. Certain aspects of the response form are nevertheless the same for all operations, and these are described here.

Just as in predicates, the response form has available the item names as free variables. If a tuple satisfies the predicate, then the response form is evaluated, and the free variables have as values the corresponding items in the tuple.

Besides the item names, there are several special free variables available to the response form. These are:

1) *TUPLE is bound to the whole tuple under consideration. The tuple is represented as a list of the tuple items in their initially defined order, as in INSERT. This list is useful when it is desirable to treat the tuple as a whole. Note: destructive list manipulating functions should not be used on the value of *TUPLE, as they may damage the underlying relation physical data structure.

2) *END is a flag for halting the operation. It is set to NIL when the operation commences. If the response form should ever set *END to a non-NIL value, then the operation will be halted at that point.

3) *RES contains the partial result of the database operation. It can be tested by the response form to determine if the operation has reached a particular point.

This description of the response form has been necessarily vague, since different operations use it in different ways. Subsequent sections give particulars and examples for each operation.

III.2.5 Complex Deletion

More complicated deletion tasks can be performed by the addition of a response form to the DELETE function. The response form, as described above, is an arbitrary S-expression which gets evaluated each time a tuple satisfies the deletion predicate.

The response form is an optional third argument to DELETE:

(DELETE relname pred rfm)

The response form has the column names of the relation available as free variables, just like the predicate. In addition, the two special variables *RES and *END can be used to control the action of DELETE.

*RES is a count of deleted tuples. When DELETE is called, *RES is initialized to zero, and incremented by one for each tuple deleted during the call to DELETE. The response form can check this variable, and can, for example, halt the deletion process after a certain number of tuples have been deleted.

To stop the deletion operation, the response form can set the *END to a non-NIL value. This variable is initially set to NIL when DELETE is called. If at any time the response form sets it non-NIL, the deletion process stops after deleting the last tuple to trigger the response form. One example -- to delete only the first 10 tuples of the PARTS table, or until a tuple with FATHER equal to JOHN is deleted:

```
(DELETE PARTS T (IF (OR (EQ *RES 10) (EQ FATHER 'JOHN))
                    (SETQ *END T)))
```

Note what happens here: each time a tuple is deleted, *RES gets incremented automatically. If *RES is equal to 10, or if the FATHER item is JOHN, then *END is set non-NIL and the deletion halts and returns the value of *RES. Else, the deletion process continues.

Note that the response form can accomplish other actions besides stopping the deletion operation. Typical uses would include printing messages if a certain type of tuple were deleted, or calling database functions to perform other database operations.

III.3 Modification

A subset of the tuples of a relation can have their values modified with a single operation. The subset is defined by a predicate on the table, and the modification by a response form. Action is as follows:

Each tuple is checked against the predicate. If it is satisfied, then the response form is evaluated, and the item values are changed if the response form modifies them. The modification continues until all tuples have been checked.

The ALISP function is called MODIFY. It takes three

arguments: the first is the relation name, the second is a predicate, and the third is a response form; all are unevaluated. Note that the response form is not optional, since it is used to re-assign the values of a tuple's elements, by using SETQ (or its relatives) on the corresponding item names. For example, to set all LID's in the PARENTS relation to negative values, use:

(MODIFY PARENTS T (SETQ LID (MINUS LID)))

Several features of this function should be noted: the predicate T, which is always satisfied, was used to select all tuples of the relation for modification. The free variable LID was re-assigned with SETQ to the negative of its initial value. The by-product of the SETQ is that the actual tuple in the relation has its LID element changed to negative.

More complicated response forms can be used, with SETQ's embedded within conditional forms so that more sophisticated modification takes place. As in the DELETE function, the *RES, *END, and *ITEM variables are available. *RES is initialized to zero,a dn incremented by one for each tuple that satisfies the predicate.

*END is a flag for ending the modification; if set non-NIL, the modification halts. *TUPLE is a list of all item values, in the initially defined relation order. Note that SETQ's should be used to change the item values, rather than using destructive list manipulating functions on *TUPLE.

MODIFY will issue errors in two cases:

1) An item value is changed to an illegal value, as specified by the relation column restrictions (see the section on Schema definition).

2) A key item value is changed. MODIFY can only be used to change non-key items. In order to change a key item, a DELETE and INSERT operation must both be performed. This is in accordance with the concept of a key as being the basic unit of information of a tuple, while the other items are its attributes.

IV. Query Operations

Here is the heart of the RDB. As was true for AI languages in general, it is the provision of a flexible query regimen which renders the RDB so attractive. For the RDB, the basic query is the selection of a subset of tuples of a single relation, using a predicate. More complicated queries may extend across several relations, and these involve the use of a response form or a special type of operation called a join.

The query operations will be described in the normal mathematical framework of the RDB query. Three algebraic operations, called restriction, projection, and join, form a complete set in that an arbitrary first-order query can be simulated using just these three operations (8). However, since the results of the query functions in ALISP are not relations (although most of them have relation-like properties), there are anomalies in the action of these functions which cannot be explained in terms of the mathematical operations. In addition, some special, extra-relational properties, such as order, are explained in later parts of this chapter.

The SELECT operation is modelled after that of the algebraic query language SEQUEL /3/. Embedding SELECT in LISP involved its own particular difficulties, which were solved by the design of predicate and response form arguments to SELECT.

IV.1 Selection

The simplest query operation is that of restriction, taking a subset of the tuples of a relation. The function SELECT is used with a predicate:

(SELECT relname pred)

Both arguments are unevaluated. The result of this function is a list of the tuples of the relation which satisfy the predicate. For example, with the PARENTS relation:

PARENTS

FATHER	MOTHER	LID
JOHN	MARY	1
JOE	SUE	2
BRAD	LOLA	3
JOE	MARY	4

---

(8)Along with the set operations union, intersection, etc. /3/

A restriction to return the liaisons which JOHN or BRAD participated in would be:

```
(SELECT PARENTS (OR (EQ FATHER 'JOHN)
                    (EQ FATHER 'BRAD) ))
```

with result:

```
((JOHN MARY 1) (BRAD LOLA 3))
```

IV.2 Response Forms and Projection

The response form is a method of embedding an arbitrary LISP S-expression within the SELECT function. By this means the power of the selection operation is increased, and its flexibility enhanced. Because the response form covers a variety of conceptually different uses within the SELECT, it will be explained and exemplified in different places throughout the rest of this chapter. The general form for the SELECT operation is the same for all uses of the response form:

```
(SELECT relname pred rfm)
```

where rfm is an optional (unevaluated) argument, a response form.

IV.2.1 Projection

Often it is desirable to return just a part of a tuple as the result of a SELECT. This operation is called projection, since only a subset of the relation columns are used. A response form is included with the SELECT for projection:

```
(SELECT relname pred rfm)
```

The response form has available the column names as free variables, in the normal manner. Continuing with a previous example, a projection operation to return only the wives of BRAD and JOHN would be:

```
(SELECT PARENTS (OR (EQ FATHER 'JOHN)
                    (EQ FATHER 'BRAD) )
                MOTHER)
```

Here the predicate selects the desired tuples, and the response form is evaluated for each, giving the MOTHER item. The results of the response form evaluations are strung together to return the final result, a list of wives:

```
('MARY LOLA)
```

To Project out more than one column of a relation, the LIST function can be employed within the response form. If wives and their associated husbands are to be projected from PARENTS, then the simplest way would be:

```
(SELECT PARENTS T (LIST FATHER MOTHER))
```

which returns the result:

```
((JOHN MARY)(JOE SUE)(BRAD LOLA)(JOE MARY))
```

One feature of response forms used in this manner should be noted. If the response form returns NIL, then it is not added to the list which SELECT returns. Thus, for example, NIL values will not be included when projecting single columns from a relation.

IV.2.2 *RES and *END.

The response form, since it is an arbitrary ALISP S-expression, can compute any function of a tuple selected by the predicate, and return it as part of the result of a SELECT. Normally, the results of the response form evaluations are strung together to make up the list returned by SELECT. However, more flexibility in forming the result is often desired. A typical situation arises when only unique values are needed in the result. A selection operation to return all the fathers in the PARENTS relation would yield two JOE's:

```
(SELECT PARENTS T FATHER)
```

with result (JOHN JOE BRAD JOE), since JOE occurs in two tuples.

The general method for dealing with situation where a different result is desired is to give the response form more control over it. This is done with the variables *RES and *END.

*RES is always set to the partial results of the SELECT function at a given point. The response form can check this variable to determine what to return next. For example, a list of fathers with no duplicates could be returned with the following selection operation:

```
(SELECT PARENTS T
        (COND ((MEMBER FATHER *RES) NIL)
              (T FATHER) ))
```

Here the response form is a COND clause which checks if the FATHER item is already in the partial result list. If it is,

then the response form returns NIL, i.e., nothing is added to the result of the SELECT. If the FATHER item is not present, it gets returned as the value of the response form, and added to the result list. *RES is updated after each evaluation of the response form.

The variable *END is available to halt the selection process at any time. If the response form sets *END to a non-NIL value, then the selection operation stops and returns the partial results accumulated to that point. One of the most common uses of *END is to halt selection after a certain number of tuples have been looked at. A SELECT function to return the first tuple of the PARENTS relation in which BRAD participates would be:

```
(SELECT PARENTS (EQ FATHER 'BRAD)
                (PROGN (SETQ *END t) *TUPLE) )
```

Here the response form sets *END to T, halting the selection operation, and returns *TUPLE, the whole tuple, as its result. Note that the result of the response form evaluation which sets *END is included in the result list of the SELECT, so that the above operation returns:

```
((BRAD LOLA 3))
```

The operation of returning the first tuple satisfying the predicate is useful enough to warrant a special pre-defined function, FETCH. This is described below in the section on pre-defined response forms.

IV.2.3 INITIAL clauses

The response form can be given complete control over the result of a SELECT function by the use of an INITIAL clause. When an INITIAL clause is given with the SELECT, the result of the response form is not automatically added to the result list of the SELECT. Rather, the response form must explicitly set the result of the SELECT by manipulating *RES.

The INITIAL clause is included after the response form:

```
(SELECT relname pred rfm (INITIAL init))
```

When the SELECT function is first entered, init is evaluated and *RES is set to this value. Subsequently the value of *RES can be changed by the response form via SETQ or its relatives. When the SELECT function exits, the value of *RES is returned as its result.

Several examples will illustrate typical uses of the INITIAL clause. The first or last (or nnt) relation tuple satisfying a

given predicate can be returned easily. For example, the first tuple in which BRAD participates in the PARENTS relation is returned by:

```
(SELECT PARENTS (EQ FATHER 'BRAD)
                (SETQ *FND T *RES *TUPLE)
                (INITIAL NIL) )
```

Here the INITIAL clause sets *RES to NIL when the SELECT begins. If there is a tuple which satisfies the predicate, the response form sets *FND to T and *RES to that tuple. Thus the SELECT halts and returns the value of *RES:

```
(BRAD SUE 3)
```

Note that this is a tuple rather than a list of a single tuple, since *RES was set to the tuple explicitly.

In a similar way, the last tuple satisfying the predicate could be returned using:

```
(SELECT PARENTS (EQ FATHER 'BRAD)
                (SETQ *RES *TUPLE)
                (INITIAL NIL) )
```

Here *RES is reset each time a tuple is found which satisfies the predicate. When all tuples have been found, the SELECT exits with the value of *RES, the last tuple found.

Finding and returning the first tuple is useful because it ends the SELECT at that point, saving the coputation involved in checking the predicate against other tuples. However, since relations are logically unordered structures, the user should not expect the same tuple to be delivered by this process each time it is carried out, if there is more than one tuple in the relation satisfying the predicate. It is true that tuples are considered by SELECT queries in an order defined by the physical storage of the relation, but this order cannot be depended upon if full data independence is to be achieved. The user may, however, specify a logical order for a query to look at tuples in a relation; this is discussed in a subsequent section on ORDER clauses.

A second common use of the INITIAL clause occurs when a SELECT is evaluated for its side-effects, rather than to return a result. In this case, the response form performs some action, but does not change the value of *RES. A simple example would be the creation of a new relation UNIONS(FATHER,MOTHER), which is a projection of the first two columns of PARENTS. One way to transfer the contents of PARENTS to this new relation would be:

```
(SELECT PARENTS T
          (INSERT UNIONS (LIST FATHER MOTHER))
          (INITIAL NIL)
```

Here the response form, an INSERT function, is evaluated for its side-effect of inserting tuples into UNIONS. The INITIAL clause sets *RES to NIL, and the response form doesn't change this, so the SELECT exits with value NIL. If PARENTS were a large relation, and the INITIAL clause was not used, then the values of the insert operation would be strung together in the result list of SELECT, wasting much computation and free storage.

IV.3 Pre-defined Response Forms

Several response forms have been pre-defined as ALISP functions for handy usage with SELECT (and PATH). If any user functions with the same name are defined, then the pre-defined functions are lost, so be careful if you intend to make use of them. All these functions start with the letter "R".

All of the pre-defined functions take one argument, and most require that the INITIAL clause also be used. All ignore NIL values from their arguments.

RFIRST, RLAST

These two return a non-NIL function of the first or last tuple, respectively, which satisfies the predicate. Note that the result itself is returned, rather than a list containing the result. An INITIAL clause must be used with these functions; if no tuple satisfies the predicate, then the INITIAL value is returned. For example, to return the first tuple in which JOHN participates, and NIL if there are none, use:

```
(SELECT PARENTS (EQ FATHER 'JOHN)
                (RFIRST *TUPLE)
                (INITIAL NIL) )
```

Note that the argument to RFIRST is evaluated, and it is the result of this evaluation which is returned; *TUPLE is the whole tuple. Any function could be computed in this argument, but usually it is the tuple or one of its items which is used.

Note that RFIRST and RLAST return their argument, rather than a list consisting of the argument.

RMAX, RMIN

These functions return the maximum and minilum, respectively, of some function over the tuples of a relation. Maximum and minimum are defined in terms of dictonary ordering for strings and literal atoms. In mixed types (strings and numbers), numbers always follow strings. For

each tuple which satisfies the predicate, the argument to RMAX or RMIN is evaluated. The minimum or maximum value is returned after all tuples have been checked.

The INITIAL clause must be included with these functions. The initial value is used in the calculation of the maximum (or minimum) result, so that if no evaluation is greater (or less) than this value, it will be returned.

As an example, the following SELECT will return the maximum LID value from PARENTS, or zero if there is none:

```
(SELECT PARENTS T (RMAX LID)
                  (INITIAL 0) )
```

RCOUNT, RSUM

These two function return the count and sum of some function of the tuples in a relation. They take a single argument, which is evaluated for each tuple which satisfies the SELECT predicate. An INITIAL clause must be included, and is used as the initial value for the sum or count.

Arguments to RSUM should always evaluate to numbers, either real or integer; if not, they are discarded. The argument to RCOUNT is only counted if it evaluates non-NIL.

As an example, to count the number of liaisons JOHN appears in, use:

```
(SELECT PARENTS (EQ FATHER 'JOHN)
                (RCOUNT LID)
                (INITIAL 0) )
```

RUNIQUE

This function is used to return a list which is a set, i.e., its elements are unique. The argument to RUNIQUE is evaluated for each tuple which satisfies the predicate, and if this result has not been obtained before, it is added to the result list. RUNIQUE will work with both lists and atoms. Example -- to return a list of all fathers from PARENTS, with no duplicates:

```
(SELECT PARENTS T (RUNIQUE FATHER) (INITIAL NIL))
```

IV.4 Joining

The join operation in the mathematical model creates a new relation which is the Cartesian product of two (other) relations. Thus, if relation A consisted of n-tuples, and relation B of m-tuples, then the Cartesian product AxB would consist of all m*n-tuples formed by adjoining the n-tuples of A with the m-tuples of B.

An outright Cartesian product is a rare occurrence in the RDB world, however. Most useful joins are equijoins, or slight variations, where tuples from two relations are joined together only if they contain a common value in one column from each relation. Such equijoins follow the semantic interpretation of the relations. For the PARENTS and OFFSPRING relations, if we wish to find which children belong to which parents, the relations should be joined using the LID column, which is the semantic link between the two. Tuples of the joined relation would be of the form:

(father mother lid child)

where the lid was the same in the original relations.

A simple technique for deriving equijoins is the embedded SELECT. A SELECT function is embedded within the response form of an outer SELECT, thus accessing two (perhaps the same) relations. This is done for the above example as follows:

```
(SELECT PARENTS T
        ((LAMBDA (LIDT)
             (SELECT OFFSPRING (EQ LID LIDT)
                     (LIST FATHER MOTHER LID CHILD)) )
         LID) )
```

Here the response form for the outer SELECT is a lambda-clause, which rebinds the value of LID to LIDT in order to prevent naming conflicts in the embedded SELECT. The embedded SELECT is invoked for each tuple in the PARENTS relation, and it selects those tuples in OFFSPRING which have the corresponding LID. The response form of the embedded SELECT returns a tuple list consisting of the FATHER and MOTHER items from PARENTS, and the corresponding LID and CHILD items from OFFSPRING. Note that the free variables of the outer SELECT (FATHER and MOTHER) are available to the embedded SELECT, as long as there are no column name conflicts.

The result of the above operation is not really a list of tuples, but rather a list of lists of tuples:

```
(((f1 m1 c11)(f1 m1 c12)...(f1 m1 c1n))
 ((f2 m2 c21)(f2 m2 c22)...(f2 m2 c2m))
 .
 .
 .          )
```

This happens because of the way SELECT's return their results. With a more complicated response form, it is possible to return a simple list of tuples from an equijoin:

```
(SELECT PARENTS T
                ((LAMBDA (LIDT)
                   (MAPC (SELECT OFFSPRING (EQ LID LIDT))
                         (LAMBDA (TP)
                             (SETQ *RES (CONS
                                (CONCONS FATHER MOTHER TP)
                                *RES) )) ))
                 LID)
          (INITIAL NIL))
```

Note that a lambda binding was used in the response form of the outer SELECT, since the embedded SELECT used the same name, LID, for one of its own item names. The equijoin is accomplished by the (EQ LID LIDT) predicate in the embedded SELECT. MAPC takes each tuple from the result list of the embedded SELECT, and adds the FATHER and MOTHER items to it. The resulting tuple is put on the result list *RES.

Even in this equijoin example, the response form seems slightly awkward and complex. Most of this complexity results from trying to return selected items in correct tuple form, i.e., as tuples made of elements from the two joined tuples. It is often more convenient (and efficient) to return a natural list result, for example:

```
(SELECT PARENTS T
                (LIST FATHER
                      ((LAMBDA (LIDT)
                         (SELECT OFFSPRING (EQ LID LIDT)
                                           CHILD) )
                       LID) ))
```

This example returns a list each of whose elements contains the children associated with a father:

```
(father (child1 child2 ... childn) )
```

Using this natural LISP data structure enables SELECT to return a more compact version of the equijoin.

None of the above examples is a particularly efficient way to structure an equijoin, since the OFFSPRING relation is checked through in an inner loop. It is often the case that embedded SELECT's can be changed to successive SELECT's, using the natural set capabilities of LISP data structures. For example, suppose that we wish to find all the children associated with a given father (who may have more than one liaison). The embedded method is:

```
(SELECT PARENTS (EQ FATHER 'JOHN)
                ((LAMBDA (LIDT)
                   (SETQ *RES (APPEND *RES
                                (SELECT OFFSPRING (EQ LID LIDT)
                                                  CHILD)) ))
                 LID)
          (INITIAL NIL))
```

This function does a complete sweep through OFFSPRING for every liaison of which JOHN is a part. Sequential SELECT's would use:

```
((LAMBDA (LIDS)
     (SELECT OFFSPRING (MEMBER LID LIDS) CHILD) )
 (SELECT PARENTS (EQ FATHER 'JOHN) LID) )
```

The PARENTS relation is checked once for all LID's which JOHN participates in, and the result is kept in the variable LIDS, which is checked in one sweep through the OFFSPRING table.

For embedded SELECT's, it should be noted that free variable name conflicts can occur, both with regard to the column names, and the special variables of the response form (*END, *RES, *TUPLE). Within the embedded SELECT, these variables are set accordingly and affect only that SELECT, and not the one containing it. It is always possible to provide a lambda-variable to resolve a naming conflict.

IV.5 PATH

The PATH function is a convenient way of expressing an equijoin, and also a much easier form to optimize. However, some of the flexibility of embedded SELECT's is sacrificed.

The format for PATH is:

(PATH rel1 collist1 rel2 collist2 pred)

where rel1 and rel2 are the two relation names, and collist1 is a column name list from rel1, collist2 from rel2. PATH returns a list of tuples formed by the equijoin of the two relations along the two sets of named columns. In the returned tuples, the tuple

items of rel1, in order, are followed by those of rel2, in order. For example, the first equijoin example done with SELECT would be:

```
(PATH PARENTS (LID) OFFSPRING (LID) T)
```

The predicate can be used to restrict the joined relation, and PATH will return only those joined tuples which satisfy the predicate. Just as in SELECT, it has available all the free variables from both relations; in the event of name conflicts, the variable takes on its value from rel2.

In the same manner as SELECT, the PATH function can use a response form and INITIAL clause. Again, the variable conflicts are resolved towards the second relation. As one example, we again find all the children of JOHN:

```
(PATH PARENTS (LID) OFFSPRING (LID)
                    (EQ FATHER 'JOHN)
                    CHILD)
```

IV.6 Ordering the Result

The results obtained from query operations on a relation have so far been returned in an undefined order, since the relations themselves are unordered. The underlying physical data structure, however, defines an order on the relations, and this is the order in which the tuples are actually returned for consideration by SELECT and PATH functions. This order should not be relied on by user programs using these functions, since it should be possible to change physical data structures without changing the way programs see the database. However, an ordering statement can be put in the query functions themseleves; this specifies the order in which tuples are returned for that query, no matter how the data structure itself is ordered. Obviously, the query will be most efficient when data structure and query ordering are the same.

The query ordering is specified by an order clause in the PATH or SELECT function:

```
(ORDER col1 col2 ... coln)
```

where col1 is the primary ordering column, col2 the secondary, etc., as in physical data structure ordering (see the section on Schema and Data Structure Definition). The order clause can come anywhere after the response form in SELECT:

```
(SELECT relname pred rfm ... (ORDER ... ) ...)
```

For example, to return a list of mothers from the PARENTS relation, ordered alphabetically by their husbands, use:

```
(SELECT PARENTS    T
                   (RUNIQUE MOTHER)
                   (ORDER FATHER))
```

Note that the ordering is done before the tuples are given to the response form. Thus functions like RFIRST and RLAST will return the first or last tuples, respectively, in the ordered relation, when an ORDER clause is used.

IV.7 Grouping the Result

The result returned by SELECT and PATH is usually a list of the results of some function over individual tuples. Often it is desirable to consider functions over groups of tuples which are related in some way.

It is possible to group tuples by common item values before they are given to the response form. The grouped columns are specified by an optional grouping clause, of the form:

```
(GROUP col1 col2 ... coln)
```

The ordering of the column names in the group clause is unimportant.

Tuples which satisfy the query predicate, and which have the same values in the grouping columns, are grouped together into a super-tuple, in which some of the elements are a set (list) of items from normal tuples. This super-tuple has the same number of elements as the normal tuple, and its elements in the grouping columns are exactly the same as other tuples. The non-grouping column elements, however, are lists of the normal tuple elements, since the super-tuple may represent more than one normal tuple. To give an example: if the grouping column for a query on the PARENTS table were FATHER, then the super-tuple for FATHER = "JOHN" would be:

```
(JOHN (wife1 wife2 ... wifen) (lid1 lid2 ... lidn) )
```

This super-tuple groups the n tuples with JOHN in the FATHER column.

The predicate in a grouping query acts on normal tuples, but the response form looks at the super-tuples. For example, to return a list of JOHN's wives, use:

```
(SELECT PARENTS    (EQ FATHER 'JOHN)
                   (RFIRST MOTHER)
                   (INITIAL NIL)
                   (GROUP FATHER) )
```

The GROUP clause, like the order clause, can come anywhere after the response form in a SELECT.

The ordering and grouping clauses can be used together in SELECT and PATH functions. The ordering is always done on the tuples before they are grouped.

V. Schema and Data Structure Definition

The schema of the database is a description (usually declarative) of the database on all levels -- the logical entities (relations and their elements for the RDB) and their interrelationships; restrictions on states of the database which are semantically and syntactically consistent; and the physical storage and access methods of the logical entities. In this section the schema description for the RDB is presented: relation definition and syntactic constraints, the physical structure declarations for relations, and methods of enforcing semantic constraints among relations (demons).

V.1 Defining Relations

The function RELATION is used to define a single relation. In its simplest form, it is:

(RELATION relname columns)

where relname is a literal atom naming the relation, and columns is a list of item names. The order of item names defines which name refers to which item of a tuple, e.g., the first item name in columns refers to the first item of a tuple. Each item name must be a literal atom, and there should be no duplicates within a relation.

Note that both arguments to RELATION are unevaluated. To define the example relation PARENTS, use:

(RELATION PARENTS (FATHER MOTHER LID))

Relation definition is dynamic: RELATION is an ALISP function, and may be used at any point in an evaluation. When a relation is defined, it is empty; if a relation with the same name was previously defined, its contents are lost by redefinition.

V.1.1 Keys

A relation as defined with RELATION above has, by default, a key consisting of all items. To specify a different key, an optional argument (unevaluated) is used, that looks like:

(KEY item1 item2 ... itemn)

where item1 thru itemn are the desired key items of the relation. An error will occur if these are not item names. The order of

item names in the KEY argument is unimportant. As an example, the PARENTS relation is:

```
(RELATION PARENTS (FATHER MOTHER KID)
                  (KEY FATHER MOTHER))
```

V.1.2 Item Restrictions

Item restrictions are declarations which define a restricted domain for a particular item in all tuples of a relation. Item restrictions correspond to type checking of abstract data types. They are useful in catching syntactic errors when tuples are inserted or modified; they can also provide information for optimization of the physical accessing routines. Item restrictions are checked on insertion and modification of any tuple in the relation, and generate an error if they are not satisfied.

Item restrictions are one simple type of syntactic consistency check that the ALISP RDB supports in a declarative form. More complex constraints on tuples entering a relation can be maintained with demons, which are a procedural consistency mechanism.

To specify item restrictions, one of the optional arguments (unevaluated) to RELATION should have the form:

```
(RESTR r1 r2 ... rn)
```

Each ri is a restriction for a single item. Any number of items can have restrictions.

Each item restriction ri is a list whose first element is the column name, and whose other elements specify the restriction. There are six types (optional arguments are separated by dashes):

i. STRING: (colname STRING n -NIL-)

The item is restricted to ALISP strings with a maximum number of characters n (n is a positive SNUM). If n is zero, then there is no maximum string length. If the optional NIL is included, then NIL is also allowed as a member of the item domain, to indicate an empty or unassigned item.

ii. SNUM: (colname SNUM -(min max)- -NIL-)

The item is restricted to ALISP integers. If the range argument is included, then the items must also be between min and max, inclusive. If NIL is included, then NIL is also allowed, to indicate an empty or unassigned item.

iii. BNUM: (colname BNUM -NIL-)
The item is restricted to ALISP real numbers. If NIL is included, then NIL is also allowed, to indicate an empty or unassigned item.

iv. LITAT: (colname LITAT n)
This is the same as the STRING declaration, except the items are literal atoms whose print names have a maximum of n characters; if n is zero, then there is no maximum. NIL is considered to be a literal atom with a print name of three characters.

v. SYMBOL: (colname SYMBOL slist)
The item is restricted to a few literal atoms. slist is a list of the literal atoms which are allowed to be item elements. There is no restriction on the length of the literal atom print names. NIL is considered to be a literal atom, and must be in slist if it is to be used in the item.

vi. LIST: (colname LIST)
The item is restricted to ALISP lists (including NIL) of any length or depth.

V.2 Physical Data Structures for Relations

At present, there are two physical storage structures for relations, one completely core-contained, the other on disk files. Before discussing these two, some general characteristics of physical data structures for relations will be reviewed.

V.2.1 Order

Relations, being sets, do not specify an ordering for their tuples. However, the physical data structure which holds the tuples imposes an order on the tuples, and this is the order in which tuples are returned from the relation for consideration by queries (unless it is overridden by an ORDER clause in the query itself). A knowledge of the way in which the tuples are to be used can help in determining the order for a data structure, and lead to increased efficiency in processing. For example, suppose it is known that tuples from the PARENTS table will usually be wanted in alphabetical order of the FATHER column; then, if the physical data structure has been ordered in this way, the tuples are passed back correctly, with no further sorting to get them in the correct order. Of course, only one ordering is possible for

a physical data structure, so that only one type of retrieval will be made efficient, namely, the one that corresponds to the data structure ordering. In many cases this presents a significant increase in efficiency over unordered data, and there is no decrease in efficiency for other queries. It is thus usually wise to specify an ordering for a data structure. If none is given, the tuples are inserted in a random way.

The order for a data structure is specified by including an order list as an argument (unevaluated) to the RELATION definition:

(ORDER coll col2 ... coln)

where the coli are column names. coll is the primary ordering column, col2 the secondary, col3 the tertiary, etc. Tuples are ordered according to the values of the primary column. If there are tuples which have the same values in the primary column, then they are ordered according to the secondary column; and if there are tuples with the same values in both primary and secondary columns, then the tertiary column orders them, etc.

Order for strings and literal atoms is defined as dictionary order. For numbers, lower values come before higher ones. Where the two types, strings and numbers, are mixed within the same column, all numbers come after the strings.

If an ordering column is restricted to SYMBOL type, then the ordering for that column is defined by the order of the symbols in the slist. Lists cannot be ordered, but come after everything else in columns with mixed types.

As an example, suppose we order the FATHER-OF relation primarily on the FATHER column, and secondarily on MOTHER:

(ORDER FATHER MOTHER)

Then typical contents of this relation might be:

FATHER-OF	FATHER	SON	MOTHER
	JOE	PETE	MAVEN
	JOHN	JOE	MARY
	JOHN	CHARLY	MATILDA

The second row comes before the third because MARY comes before MATILDA in dictionary order. If MOTHER were declared the primary ordering column, then the first row would become the last row.

V.2.2 Factors

A factor for a data structure is like an order, in that it groups together tuples with the same values for certain items. However, there is no order necessarily imposed within a grouping

or between groupings. Factoring, since it groups together tuples with common values, can eliminate redundant storage of these common values by factoring them out from each tuple in the group, and saving only the unfactored items in the data structure. Factoring is thus used to save space, especially with large disk file data structures.

Whether factoring is useful or not for a given relation depends on the number of tuples which can be expected to have common values for the same items, and the way in which those tuples are to be retrieved. Certain types of relations lend themselves to factoring, where the information falls naturally into large groupings. For example, let us extend the PARENTS relation to include the town in which the people live:

PARENTS(FATHER,MOTHER,LID,TOWN)

Now, it is likely that there will be a large number of parents in the same town. If the town name were listed just once for each town, it would save a great deal of space over repeating it for each tuple belonging to the town. The TOWN column would thus be a likely candidate for factoring. Note that factoring out the TOWN column forces the data structure to group together all those tuples belonging to the same town. If processing usually proceeds town by town, then this might even speed things up; but if it proceeds in some other manner, say by father's name regardless of town, then the factoring will not give efficient performance, even though it will save storage space.

Factors are specified by including a factor list as an argument (unevaluated) to the RELATION definition:

(FACTOR col1 col2 ... coln)

The order of column names does not matter. Tuples which match in all the factor items are grouped together.

If an ordering is given along with a factor for the physical data structure of a relation, then the order must include the factored items. In the PARENTS example, order would be primarily by TOWN, and then by any other column.

V.2.3 Access Paths

Access paths are auxiliary data structures for a relation which allow certain types of tuples to be retrieved more rapidly. Two common sorts of access paths are inversions and links. Inversions act only on a single relation, while links connect two relations.

Inversions are indices of single columns of a relation. They enable rapid retrieval of tuples when the retrieval specifies a value for the inversion column. For example, we might invert the SON column of the FATHER-OF table. Then any retrieval which specified the SON column, say SON = "CHARLY", would have fast access to all those tuples whose second element was CHARLY. Inversions are useful if a large number of retrievals will ask for tuples with a constant value in the inversion column. The trade-off is that they take more space, and insertions and deletions for the relation are slower.

Links connect two tables, based on common values in one column of each table. The PARENTS and OFFSPRING tables furnish a good example here. They are semantically linked by the column LID in both tables, since offspring are tied to their parents by having the same LID value. It seems natural to establish a link between the two tables, connecting the tuple in PARENTS to all tuples in OFFSPRING with the same LID value. A retrieval which asks, say, for all the offspring of a given father, will then become efficient, since offspring are directly linked to their parents, and the whole OFFSPRING relation does not have to be searched. As in inversions, the trade-off is in increased space requirements, and increased time to insert and delete tuples.

None of the physical data structures implemented at the present time use either inversions or links, so this presentation on access paths is perhaps more informative than useful. However, future development of the ALISP RDB will certainly include access paths for data structures.

V.2.4 Data Structure Types

There are two physical data structures currently available for tables: lists and coded files. The list type is the default if no type is specified.

i. List structure

Relations are encoded as variable length lists of tuples, and are completely core-contained. List structures cannot be factored. To define a relation with a list structure, no further arguments to RELATION are necessary, since this is the default.

In general, it is better to use some order for list data structures, especially if the order and key items are the same, since this speeds up processing of insertions and deletions.

ii. Coded file structure (CFILE)

Coded files are a disk storage structure. This organization permits an arbitrary amount of data to be kept in a relation parameterized in this way. Unlike list structures, the CFILE must be factored on at least one column. The record structure of the CFILE is defined by the factoring, in that each record contains all tuples with the same values in the factor columns.

Besides being factored, CFILE tables must be ordered and keyed on at least the factor columns (ordering and keying on other columns is optional). This makes a large class of CFILE retrievals relatively efficient.

A table is given a CFILE structure by including the following argument (unevaluated) with the RELATION function:

(CFILE fname n)

where fname is the name of the indirect access permanent file which holds the table tuples. n is an optional integer used when several relations are to share the same CFILE. Each relation must be factored in the same way, i.e., each relation must have the same number of factor columns, and each column should be of the same type and in the same order in each relation. Each relation to share the coded file should be given a successive integer n, starting with 1.

Since the files exist independently of the relation definition, they must be manipulated by other operators which initialize, update, and restore them. These operators are described in more detail in section IX.

V.3 Demons

In a complex database environment, where perhaps many users are updating the same information, it can often occur that a mistake is made, and an inconsistent representation is formed. In the PARENTS relation, for example, a man may be listed as being married to two different women. Queries to this relation will still work, but their results may not be what is intended, since the user assumes that the relation is semantically correct. Knowing when the database is inconsistent, and finding where it is inconsistent, are non-trivial tasks in a large database.

To some extent the consistency problem is alleviated by correct use of keys, normal forms, and column restrictions. However, as the PARENTS inconsistency above shows, it is not hard to produce situations where everything is correct according to

these restrictions, and yet the semantics of the information are violated.

Demons are procedures which are useful in automatically maintaining semantic consistency of the database. A demon watches over a particular relation and a particular operation (insert, delete, modify). Demons consist of two parts: a trigger and an action. The trigger is a predicate that tests to see if the operation involved a particular type of tuple. The action is a response form which is evaluated when the predicate triggers.

Demons are defined when a relation is created. The demon declarations are optional (unevaluated) argument to RELATION:

(RELATION relname columns ... (demontype dpred daction) ...)

where demontype is one of IDEMON, DDEMON, or MDEMON (for the three database operations). dpred is the demon predicate, and daction is an arbitrary S-expression which is evaluated when the predicate triggers. There can be multiple demon declaration clauses, of any type, within a single RELATION function.

Testing of the trigger predicate is dependent on the operation associated with the demon. For insertion demons, the predicate is evaluated after each new tuple is inserted into the relation, and it tests the newly inserted tuple. If it retuns non-NIL, the insertion demon's response form is evaluated, with the new tuple's items bound to the item names. As an example, here is an insertion demon which checks for and prints BILL's liaisons each time he is inserted into the PARENTS relation:

```
(IDEMON (EQ FATHER 'BILL)
        (SELECT PARENTS (EQ FATHER 'BILL)
                        (PRINT *TUPLE)
                        (INITIAL NIL) ))
```

For deletion and modification, a demon's predicate is evaluated for each deleted or modified tuple, just after it is deleted or modified. Suppose we wish to delete children of parents when we delete their liaison from the PARENTS relation. The following demon will accomplish this:

```
(DDEMON T ((LAMBDA (LIDT)
             (DELETE OFFSPRING (EQ LID LIDT)))
           LID))
```

Note that a demon's action is completely general, since any LISP expression can be evaluated.

It would be wise to point out some of the inherent shortcomings of demons at this point. Since the action of demons is quite arbitrary, it is possible to unknowingly create demons

which try to maintain contradictory consistency constraints, and when this occurs, infinite looping may result as the demons alternately are invoked, each undoing the action of the other. Because of the procedural nature of the demons, it is impossible to check for cases like this automatically, and the user must insure the correctness of his demons.

A better solution to database consistency checking would be predicate calculus-type declarations which were consistency constraints on relations. Such constraints would be easy for the user to write and understand. The system would have an added burden of interpreting and enforcing the constraints. Design of this part of the system would be a major useful project for the RDB.

Another objection to automatic constraint checking in general (and hence demons) is that several operations on the database may produce intermediate states of the database which are inconsistent, but the final result is consistent. Since demons are applied at the end of each operation on the database, it may be impossible to achieve certain consistent states of the database, if it requires going through intermediate inconsistent states. (9)

Those who are familiar with the CONNIVER language will recognize demons as serving the same purpose as IF-ADDED and IF-REMOVED proceedures in that language. The predicate in a demon corresponds to the pattern in the IF function, and both result in arbitrary actions. An important difference lies in the scope of view of the predicate triggerring mechanism; for IF procedures, it is the entire database, while the RDB demons are tied to a particular relation.

---

(9)Some RDB's handle this problem by temporarily suspending all consistency checking over a group of operations, and then re-applying the constraints once the whole group has been completed.

VI. Views

As was discussed in the first part of this document, a database represents information with primitive entities and relationships between these entities. It is the data structure or structures supported by the database which define the basic units of information representation. In the RDB, tuples, grouped into relations, are the basic units. What meaning we give to a particular relation is the basic, or explicit, information represented by the relation; picking up a previous example, the OFFSPRING relation unites children and the liaisons from which they sprang. Any other meaning we find in the database relations is called implicit, or derived information. We could define grandchildren, for example, using both the PARENTS and OFFSPRING relations; but there is no explicit grandchildren relation.

It is easy to see that the choice of these base relations makes certain types of information easier to extract than others. A simple query will yield the children of a given liaison, but a more complex query is needed to find all the grandchildren of a given person. Implicit information, which is not represented simply by a relation, will in general require a more complex query to extract. A user, intent on extracting a particular type of information from the database, may not wish to deal with the complexity of the base relations which represent it. If he wishes only to know about grandparent-grandchild linkages, it seems senseless to make him worry about liaisons and offspring relations. What is needed is a method for making the implicit grandchildren information explicit, i.e., by forming a new relation.

The mechanism for forming new relations from the initially defined base relations is called a view. A view is simply a relation derived from other relations. For example, we might define a view, based on the OFFSPRING and PARENTS relations, which relates grandchildren to their grandparents. The normal database operaions can then be conducted against the view, rather than against the base relations.(10) Many different views involving the same base relations can exist at the same time; views may also be written in terms of other views.(11)

In an algebraic query language, views can be defined by a query operation involving other relations (which may themselves be views). Thus any (first-order derivable) implicit information contained in the database can be made explicit by the use of a suitably defined view. In this implementation of the ALISP RDB,

---

(10)Not all operations can be performed in the same way against a view. View operations are explained more fully later on.
(11)A view definition may not be recursive, that is, may not refer back to itself. Although query operations are first-order complete, this does not include recursive definitions.

however, views are restricted to using only the SELECT operation. Even with this restrction views are a very powerful mechanism; in the section on deductive mechanisms we show how to define AI database "contexts" using the view mechanism.

Views are applicable to other tasks besides making some queries easier to write. In databases shared by many users, views can provide a protection mechanism, restricting a user's access to a subset of the whole database. For AI applications, it is the deductive aspect of the view which merits the most consideration. The following section explains how to define views, and the next explores the view as a deductive mechanism for the RDB.

VI.1 Defining Views

A view is defined in the same manner as a SELECT query, with a few minor modifications. The general form of a view definition is:

```
(VIEW vname relname pred rfn
      columns initial order group)
```

Anything after pred is optional. No arguments are evaluated.

vname is the name of the view; like relation names, it must be a non-NIL literal atom. Any view or relation with the same name is destroyed.

relname is the name of the relation from which the view is formed. It need not be defined at the time the view is defined; however, it must be defined when the view is first used.

pred is a predicate which subsets the relation relname.

rfn is a response form (optional) which gets evaluated for each tuple which matches the predicate. *END and *RPS are available as usual.

columns is an optional list of columns for the view relation. It is of the form:

(COLUMNS C1 C2 C3 ... Cn)

where C1 thru Cn are column names. If this list is present, these names will be used as the column names for the view relation. If it is not, the columns names of

relname will be used.

initial, group, and order are arguments which have the same form and result as the corresponding SELECT arguments.

The action of views will be illustrated by several examples. A new base relation called HOFFSPRING will be the target of these examples:

HOFFSPRING(HID,LID,CHILD)

where HID is a household identifier. This relaion has information about children from many different households; each household has its own unique HID. Typically HOFFSPRING will look like:

HOFFSPRING

HID	LID	CHILD
1	1	JOE
1	1	JOHN
1	2	MARY
2	1	SUE
2	2	SAM
2	3	JOSY

Thus in household 1, there are two children from liaison 1, and one child from liaison 2, etc.

1. Suppose we are interested only in the first three households. Then a view subsetting HOFFSPRING in an appropriate manner would be:

(VIEW HH3 HOFFSPRING (LESSP HID 4))

Now, a query using HH3 will be the same as a query using HOFFSPRING, except that only tuples from the first three households will be considered. This is a restriction on the HOFFSPRING relation.

2. Suppose we wish to look at only male children and their households, and don't care about females of the liaisons. A relation constructed accordingly from HOFFSPRING would be:

MALECHILDREN

HID	MCHILD
1	JOE
1	JOHN
2	SAM

This relation could be defined as a view:

```
(VIEW MALECHILDREN HOFFSPRING
                   (MEMBER CHILD MALENAMES)
                   (LIST HID CHILD)
                   (COLUMNS HID MCHILD) )
```

MALENAMES is a list of names of males. The response form (LIST HID CHILD) forms the tuples for the MALECHILDREN view, and the COLUMNS form renames the columns.

3. We want to identify the number of children in each household. A suitable relation would be:

HKIDS

HID	KIDS
1	3
2	3

since there are 3 children in the first and second households. This relation could be defined as the view:

```
(VIEW HKIDS HOFFSPRING T
                   (LIST HID (LENGTH CHILD))
                   (COLUMNS HID KIDS)
                   (GROUP HID))
```

Here the GROUP clause groups the children together by household in the same way as a SELECT would, so that the value of CHILD is a list of children associated with a household.

Any SELECT query can effectively be defined as a view, even embedded SELECT's, so that the view mechanism is quite general. Currently views are implemented as query modifications; the implications of this and several other aspects of views are discussed in the next sections.

The function VIEWS will define a number of views, in the same format as the RELATIONS function.

VI.2 Operations on Views

Since views are built on base relations, operations on views (insert, delete, modify) must be implemented as operations to the underlying base relations. Unfortunately there seems to be no simple, consistent way to do this. The ALISP RDB does not solve this problem, but does provide for certain simple operations with respect to views.

Since views are defined with respect to a single underlying relation, all operations affect that underlying relation. (12) For the purposes of deletion and modification, the view predicate is used to subset the relation. These operations will only affect tuples which match the view's predicate as well as the operation's predicate.(13) Insertion into a view is the same as insertion into the base relation, except that the inserted tuple must satisfy the view's predicate.

In views which do a simple subsetting of a base relation, this definition of the operations is satisfactory. For more complicated views, the user must be aware of the structure of the underlying relations if he wishes to change the contents of the view correctly.

VI.3 Views as a Deductive Mechanism

Current AI languages usually offer some mechanism for retrieving implicit information from their databases. Such mechanisms customarily are automatically invoked, that is, the user doesn't worry about whether the information he is looking for is explicitly represented in the database or is represented by a deductive procedure. A retrieval request will automatically trigger the correct deductive procedure if the information is represented implicitly, just as a query on a view automatically invokes the view definition.

There are two broad categories within which automatically invoked deductive mchanisms fall. The first is a very powerful, very general mechanism, usually any procedure which can be written in the host language. The retrieval request triggers this user-supplied procedure, which then returns the appropriate data. While such a mechanism can do any needed deduction in an efficient manner, it places a large burden on the programmer. The concept of data-independence is destroyed, since efficiency can only be achieved if the programmer knows something about the storage characteristics of the data, and changing these will necessitate a rewrite of the deductive procedure.

The second category is exemplified by PLANNER's consequent theorems. A consequent theorem is of the form:

A → B

---

(12)A view defined in terms of another view will eventually have a base relation at the end of the view chain.
(13)A view defined in terms of another view will have both view predicates and'ed together to check the inserted tuple.

which says that, if you are looking for information represented by B, and B isn't in the database, then look for A, since the existence of A implies the existence of B. B represents implicit information, since it is implied by A.

PLANNER's consequent theorems were a good idea, since they gave the user the ability to deduce implicit information with simple, declarative statements. They are also a very powerful mechanism: while it is not readily apparent, consequent theorems are a first-order complete deductive mechanism /4/. Also, the theorems would fit into a data-independent environment, since they contain no reference to the physical characteristics of the data.

Since the consequent theorems operate over the finite domain of the database, it is not necessary to use a general theorem prover in order to find A; a simple strategy would be to check each tuple in the database to see if it matched the conditions of the theorem. Such a search, being finite, would always terminate, although for a large database it is clearly impractical. In practice, a smarter strategy uses the conditions on A to restrict the search space. This works fine for simple deductions, but when chains of deductions are encountered:

A → B
B → C
C → D

the search space will grow exponentially with the length of the chain. The proposed remedy is to let the user write his own search strategy, i.e., place the deductive mechanism back into the first category /17/.

A better remedy would be to stick with a declarative deduction mechanism, but make it more efficient, even for chains of deductions. This is exactly the kind of facility that views in a relational database can provide. The relational structure gives syntactic and semantic clues to the system to optimize search. Views are defined in a declarative manner by an algebraic query; this query can be optimized at several points: physical storage, syntactic and semantic levels. Syntactic and physical storage optimization will be discussed in a later section. Semantic optimization involves semantic clues, supplied by the programmer, for cutting down the search space. (14)

---

(14) In an experimental system called DEDUCE, these clues take the form of heuristics and preferences which guide the search strategy. The effectiveness of these techniques is demonstrated in a few examples, and although there are many issues to be resolved, it seems likely that this will be a fruitful line of research/4/.

The key question here is whether opimization at the syntactic and physical storage levels can enable declarative queries to approach the efficiency of hand-programmed procedures, or whether additional knowledge about the semantics of the representation must be made available. Implementors of stand-alone query languages such as SEQUEL gamble that syntactic strategies are sufficient. Some reasonable work in optimization at these levels has been done, and it appears that they may be right. One problem that optimizers at the syntactic level face is only knowing about the declared database structure. Any other semantic constraints which are present but undeclared could be used by a programmer coding up his own routine, but obviously wouldn't be available to the optimizer. (15)

There is another technique which may throw the balance completely in favor of the declarative deduction mechanism in an RDB. Since the result of a query is a relation, and views are defined as queries, we may represent the view explicitly by a stored relation, rather than implicitly as a query. Overhead is increased for insertion and deletion operations to the underlying base relations, and more storage is required; but no computation is required to "deduce" the information. By judicious use of this technique, an arbitrary balance between explicit, redundant storage and deductive derivation of information could be achieved.

Finally, there are some severe limitations in a first-order complete deductive mechanism. In particular, recursive definitions of sets are not part of the predicate calculus. Some sets which are easily defined recursively are a mess to specify in a first-order language, while others simply can't be defined. In these cases, the programmer must write his own routines in the host language, using database queries as primitives.

Currently, this implementation does not provide for any semantic optimization, and there is limited syntactic and physical storage optimization. As an escape hatch, an arbitrary function can be evaluated as the response form of a view, and the programmer can construct arbitrary deductive procedures of his own. Future research will concentrate on providing better optimization techniques to relieve the programmer of this burden. No solution to the limitations of first-order definitions is proposed, however.

---

(15)The heuristics and preferences of DEDUCE can provide some of this additional information to the optimizer.

VII. Packages aka Contexts

Contexts are a peculiarity of AI language databases. They satisfy an aching need of many AI projects: the ability to make incremental changes to the database, and be able to backtrack to previous states of the database, automatically removing these changes. Such a facility is useful, for example, in game-playing, problem-solving and model-building activities.

The backtrack points in early contextual databases were kept for every incremental change (addition or deletion of single tuples). CONNIVER introduced the useful innovation of grouping together changes into sets called layers /17/. Backtrack points were established only at the beginning of each layer; all changes in the layer were undone as a complete set.

A context consisted of a hierarchically ordered set of layers. Model-building programs would typically grow a new layer onto the current context in order to make some local changes to the database. If the results of the changes were not satisfactory, the new layer could simply be removed from the context, and the database would be as it was initially.

Another way to look at contexts is to regard each layer as a small packet of information. A context is a database formed by combining the information in some set of layers. This is the approach taken in an implementation of partitioned semantic nets /13/, and which we follow here. The name "packet" is chosen to refer to a "packet of information" in the database. Sets of packets, forming a complete database, are referred to as a "package".

The purpose of this chapter is to show how a partitioning of the RDB along the lines of contexts can be easily accomplished with the view mechanism. While the design of all AI language databases has provided a single type of partitioning, this is not true of the RDB. It is possible to model many different types of database partitioning using the view mechanism and base relations. The particular example of this chapter was chosen for its utility and simplicity of implementation. Projects which need special types of database partitioning should be able to design and implement them with a minimum of programming effort. Finally, the flexibility of physical data structure definition available to the RDB should be able to make the implementation of partitioning reasonably efficient.

The relations:

LIAISONS(LID,PERSON,LTYPE)

OFFSPRING(LID,CHILD)

will be used as an example in this section. These relations

represent liaisons between people by linking them through common liaison identifiers (LID's). LTYPE is the type of liaison. If JOHN and MARY were married, and JOHN had been divorced from JOSY, then we would have:

LIAISONS

LID	PERSON	LTYPE
1	JOHN	DIVORCED
1	JOSY	DIVORCED
2	JOHN	MARRIED
2	MARY	MARRIED

OFFSPRING

LID	CHILD
1	SAM
2	SUE
2	ARCHY

SAM is the son of JOHN and JOSY, while SUE and ARCHY are children of JOHN and MARY.

VII.1 Packets

Conceptually, a packet is a (usually small) set of database tuples. These tuples need have no connection between them, other than being in the same packet. They may come from the same relation, or from many different relations. It is perhaps easiest to think of a packet as a small database, an arbitrary subset of the whole database.

Packets can overlap, so that the same tuple can be contained in more than one packet. This is in keeping with the meaning of packets as "packets of information", or tuples which are semantically grouped. Any single tuple could belong to many such groupings, according to the user's ideas about what information belongs together.

Let us illustrate these concepts with several concrete examples using the LIAISONS and OFFSPRING relations. A packet might hold all tuples belonging to the family of JOHN and JOSY. Call this packet P1. P1 can be represented as a region enclosing the appropriate tuples from the OFFSPRING and LIAISONS relations, as in figure 1.

Another packet, called P2, might contain tuples relating to married people. This packet (along with P1) is shown diagrammatically in figure 2.

Packet names are used to refer to the set of tuples in a packet. These names must be either non-gensym literal atoms or SNUM's. Functions for creating new packets, and adding tuples to packets, are to be found in subsequent sections.

Figure 1

Packet P1

LIAISONS

LID	PERSON	L'TYPE
1	JOHN	DIVORCED
1	JOSY	DIVORCED
2	JOHN	MARRIED
2	MARY	MARRIED

OFFSPRING

LID	CHILD
1	SAM
2	SUE
2	ARCHY

Figure 2

Packets P1 and P2

LIAISONS

LID	PERSON	L'TYPE
1	JOHN	DIVORCED
1	JOSY	DIVORCED
2	JOHN	MARRIED
2	MARY	MARRIED

P1

P2

OFFSPRING

LID	CHILD
1	SAM
2	SUE
2	ARCHY

VII.2 Packages

Packets help to organize semantically related tuples, forming a subset of the database. The union of several packets can be used to form larger subsets of the database. Such a union is called a package. If packets are thought of as units of information, then a package is a world view composed of many such pieces. To the user, a package looks exactly like a database, except that he has a very flexible facility for including or excluding packets of information from the package. Any database operations done to a packaged database will see only the subset of tuples in the package.

A package is specified by a list of packet names, e.g.,

(P1 P2)

is a package consisting of all tuples in P1 and P2. Since a package is the union of packets, the order of the packet names is unimportant.

While it is possible to form packages from arbitrary combinations of packets, in a model-building environment there is often a hierarchical structure to packages. A new package is created by adding a packet to a parent package. The resulting structure can be diagrammed as in figure 3.(16) Arrows indicate the direction in which further database tuples can be found.

Figure 3

Packages

P1 (P1

P2 (P2 P1) P3 (P3 P1)

P4 (P4 P3 P2 P1)

P5 (P5 P4 P3 P2 P1)

---

(16)This representation is borrowed from /13/.

Package (P1) contains only packet P1. Package (P2 P1) contains all tuples in either P2 or P1, but not in P3, P4, or P5. From package (P5 P4 P3 P2 P1) , all tuples in the packets are part of the database.

Functions are provided to create packages in this fashion from packets. These functions will create partially ordered hierarchies of packages, as diagrammed. Note however that this is just one convenient way of using packages. More general network structures of packets can also be easily created when necessary.

At any given time, database operations see only that part of the database contained in the current package. This package is the value of the atom PACKAGE, in the form of a list of packet names. The current package is changed by changing the value of PACKAGE.

VII.3 Packaging Relations

Not all relations need participate in the database subsetting defined by packages. A relation must be declared to be a packaged relation when it is created for packages to have any effect on it. The RDB thus consists of two different sets of relations: those which act normally under database operations, and those which are subsetted by packets.

A packaged relation is defined in the same way as an ordinary relation, with the addition of the atom PACKAGE as one of the optional arguments. LIAISONS defined as a packaged relation would be:

```
(RELATION LIAISONS (LID PERSON LTYPE)
                   (KEY LID PERSON)
                   (ORDER LID)
                   PACKAGE)
```

Actually, a packaged relation defines two things: an underlying base relation which holds the tuples, and a view on that base relation which implements the package construct. The name of the base relation has an asterisk prefix. In this case, defining LIAISONS as a packaged relation causes the definition of the underlying base relation:

```
(RELATION *LIAISONS (LID PERSON LTYPE *PAK)
                    (KEY LID PERSON *PAK)
                    (ORDER LID *PAK) )
```

where the value of *PAK is a package name. A view on LIAISONS is also defined:

```
(VIEW LIAISONS
     *LIAISONS
          (MEMBER *PAK PACKAGE) )
```

All database operations which use the package should be made with respect to LIAISONS rather than *LIAISONS. The user may change *LIAISONS at his own peril.

VII.4 Packaged Relation Operations

The basic query operations are the same as for normal relations, SELECT and PATH. When applied to packaged relations, these functions only look at the set of tuples in the current package. This mechanism is implemented by views, as explained in the previous section. The atom *PAK is available as a free variable in these operations; its value is the packet in which the tuple resides.

Tuples are added to packaged relations with INSERT. The tuple to be added should have an extra item, a packet name, tacked onto the right. This is the packet in which the tuple will reside. To place the tuple in more than one packet, one insertion must be done for each packet. For example, the tuple (1 JOHN MARRIED) can be inserted into LIAISONS as part of packet P1 with:

```
(INSERT LIAISONS '(1 JOHN MARRIED P1))
```

Commonly, tuples are added to the packet which is the first one on the packet list of PACKAGE. The special function ADD has been provided for this purpose. If the value of PACKAGE is (P1 P2), then the following two operations are equivalent:

```
(INSERT LIAISONS '(1 JOHN MARRIED P2))

(ADD    LIAISONS '(1 JOHN MARRIED))
```

Packets must be explicitly removed from the database; removing their names from the current package does <u>not</u> destroy them. The function for deleting packets is:

```
(KILLPAK packetlist r1 r2 ... rn)
```

where <u>packetlist</u> is a list of packet names to be deleted, and <u>r1</u> thru <u>rn</u> are (unevaluated) relation names from which all tuples in the packets will be deleted. Note that the user must keep track of which packets reside on which relations if he wants to get rid of them eventually.

DELETE and MODIFY may be used to delete and modify tuples in a packaged relation. They will only look at tuples in the current package. By setting the current package to a single packet, it is possible to delete or modify only tuples in that packet.

VII.5 Manipulating Packages

Packages are just lists of packet names. They can be manipulated by the regular ALISP list manipulating functions. To change the current package, the value of PACKAGE is reset to a new list. Some new functions are provided to make frequent operations easier. PACKAGE is initially NIL when ALISP is loaded.

(PUSHPAK -packet-)

packet is an optional (unevaluated) packet name. This function adds packet to the beginning of the current package. If packet is not present, the system provides a new packet name in the form of an integer. This integer is the value of the atom PACKNUM, and is incremented on each call to PUSHPAK. PACKNUM is saved with the database, so that it will yield a new packet name whenever the database is loaded. PACKNUM can always be set by the user for special purposes.
PUSHPAK returns the new packet name as a result.

(POPPAK)

This function pops the first packet name from the package list, and returns it as its result.

To get a new packet name for your own use, without adding it onto the beginning of PACKAGE, use:

(GENPAK)

which returns the next packet integer and increments PACKNUM.

VIII. Optilization and compilation

There is a simple algorithm for returning the results of any SELECT or PATH query. This algorithm involves an exhaustive search: check each tuple of the relation (or cross-product of two relations for PATH) against the predicate, and evaluaate the response form if it returns non-NIL. Optimization is the search for better algorithms for certain classes of queries and underlying physical data structures.(17)

VIII.1 Key Optimization

Queries which ask for a specific tuple by key can be optimized over a variety of physical data structures. Arrays are used with a dense numeric key; binary trees, indexed sequential files, hash tables, and other physical structures are used wtih any key type. Each of these physical storage structures has a fast access method for retrieving aa single tuple with a given key.

Key optimization occurs when physical storage is one of the above types, and a SELECT predicate has the correct form as follows: if K1 ... Kn are key items in the relation, then the query predicate must be:

(AND ... (EQ K1 S1) ... (EQ Kn Sn) ...)

where S1 ... Sn are any S-expressions which do not involve item names. Other conditions can be specified by additional clauses to AND, but the EQ clauses must be there for all key items. Essentially, this predicate specifies the single tuple whose key is S1 ... Sn.

Currently, key optimization is not employed, since there are no physical data structures which have the key access property.

VIII.2 Order Optimization

Queries which ask for more than one tuple can often be optimized to look at only a part of the query relation. The physical data structure must be ordered in this case. The query predicate is parsed to determine order bounds in searching the

---

(17)Complex, embedded queries can also be optimized by query restructuring, without looking at the physical storage characteristics of relations. This type of optimization is seen as less useful for the ALISP RDB, where queries are mainly simple SELECT's or PATH's.

physical data structure.

Suppose the relation

PARENTS(FATHER,MOTHER,LID,TOWN)

is ordered primarily by TOWN. An order optimization can be made for queries like:

```
(SELECT PARENTS
        (AND (OR (EQ TOWN 'AMHERST)
                 (EQ TOWN 'GRANBY) )
             (LESSP LID 6) ))
```

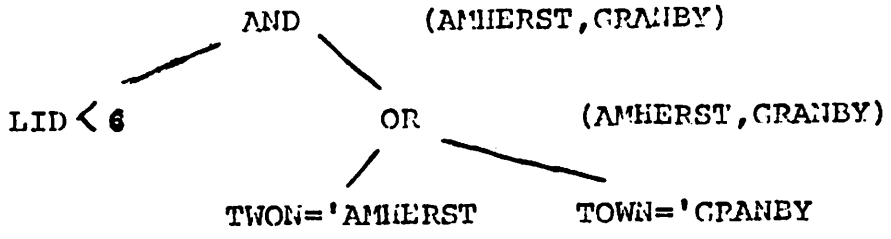
A parse tree of the predicate is formed with the nodes of the tree being boolean predicates and the leaves primitive predicates. At each OR junction, order values are added to give a larger order range; while at AND nodes they are intersected to give a smaller range (see diagram). The relation need only be searched from TOWN="AMHERST" to TOWN="GRANBY". Primitive predicates used in the parsing process are MEMBER, EQ, LESSP, and GREATERP.

Access to any physical physical data structure which has an ordering can be optimized in this way (hash tables and unordered lists are the only contemplated physical structures which do not have an order). Order optimization currently works particularly well with CFILE data structures, since restricting the order range will restrict the number of buffer transfers needed to complete a query.

VIII.3 Algorithm Selection

Order optimization also occurs in selecting an algorithm for

Parse tree for SELECT predicate



SELECT's which use an ORDER clause. If the result order as specified by the ORDER caluse is the same as (or subsumed by) the physical data structure order, then tuples are returned from the physical data structure without sorting. For those queries where the order specification is not the same as the physical ordering, the tuples are first selected by the predicate, then sorted according to the ORDER clause before being handed to the response form.

The PATH query (equijoin) can be optimized by choosing a suitable algorithm, depending on the charactersitics of physical storage for the two relations. This is not currently implemented.

VIII.4 Compilation

Optimization of queries makes the RDB efficient as a data retrieval mechanism. However, optimization involves some investment in processing time, and, at least for queries on relations with few tuples, this overhead can swamp the benefits of efficient execution. Also, if the query is excuted more than once, the optimization must be performed each time, even though the query and the relation's physical storage do not change.

In order to bypass optimization overhead during execution, we can try to do as much as possible during compilation. At compilation time, however, we do not know the structure of dynamically defined relations, and query predicates may also not be completely defined. Thus complete otpimization cannot always be performed at compilation, but the process can be carried forward as far as possible. For example, the previous PARENTS query predicate could be parsed completely to yield a min-max bound for order optimization at compile time. Partial parsing can be performed on predicates which have undefined expressions at compile time:

```
(AND (OR (EQ TOWN T1)
         (EQ TOWN T2) )
     (LESSP LID 6) )
```

Since T1 and T2 are not know at compile time, a runtime check for the relative values of T1 and T2 is made before an ordering bound is arrived at:

```
T1  T2 - (T2,T1)
T2  T1 - (T1,T2)
```

The work of constructing and analyzing the parse tree would be done only once at compile time, leaving a simple runtime check. Note that, in general, the relation must be declared at compile tile in order to optimize a query; if it were not, we could not

even tell, in the above example, which free variables refer to item names.

Considering relations as abstract data types, compilation can achieve their efficient implementation as normal LISP data structures. Consider a standard commercial example, a relation between employees and their salaries:

EMP

NAME	SALARY
JOE	13,000
JOHN	12,000
BRAD	2,000

If the physical data structure were a binary tree, then queries asking for the salary of a particular employee would compile to a call to a LISP binary tree search routine. Now suppose the NAME domain is replaced by a dense set of positive integers, i.e., employee id's. Then a good physical data structure would be an array whose index was the employee id. A query to find an employee's salary would compile to a simple array element fetch. Thus, the compiler would actually convert the query into one of a library of search reoutines for the different storage methods, and runtime efficiency would be achieved without the loss of data independence.

Compilation of queries is not currently done on the ALISP RDB. This is another major project that needs doing.

IX. RDB Auxiliaries

The functions described in this section are utilities used in setting up and printing out various parts of the database, as well as manipulating files and other housekeeping chores.

IX.1 Initialization

The database functions are already loaded with the ALISP system, unless some user-defined function with the same name has overridden them.

A database is given a name so that it can be referred to by saving and restoring functions. The current database name is the value of the atom DATABASE, and is initially RDB. The user should not modify DATABASE, except with the functions provided below. Also, the property list of the database name contains important information used by the database primitives, and should not be disturbed.

The database name can be changed, and a new database initialized, by evaluating:

(NEWDB -dbname-)

where dbname is an optional literal atom naming the database. If this argument is not included, RDB is assumed. Any information in the previous database is lost (except for coded file relations), and the new database is intially empty.

IX.2 Saving a Database

A database can be saved on a single ALISP permanent file by evaluating:

(SAVEDB -fname-)

where fname is the name of the file on which the database will be saved. If fname is omitted, then the file name will be the same as the database name.

Before saving a database, all active files should be closed (see below on files).

If fname is not already a permanent ALISP file in the user's catalog, SAVEDB will initialize it as such.

IX.3 Restoring a Database

A database can be restored from a file where it was previously saved by evaluating:

(OLDDB fname)

where fname is the name of the file on which the database had been saved. The condition of the database is exactly that at which it was saved. Thus, many different copies of the same database can be stored, for such purposes as backing up in case of a mistake.

IX.4 Listing Relations

To get a listing of some or all relation definitions in the current database, use:

(RELATIONS tab1 tab2 ... tabn)

where the tabi are relation or view names. If no relation names are given, then all definitions are listed.

The contents of a relation can be listed in a nice format by:

(PRINREL relname -pred-)

where pred is an optional predicate which picks those tuples to print. If pred is omitted, all tuples are printed.

IX.5 Files

Besides the relation definitions and core-contained relation datas, a database associates coded files with those relations defined to have a CFILE physical data structure. These files are saved and restored independent of the database itself.

Opening of the files is automatic: when a database function references a relation defined as a CFILE, the appropriate file (given by the relation definition) is opened if it exists, or created and initialized if it doesn't. The user should be careful not to have any file in his catalog with the same name as given in a relation definition, since an error will occur if it has not been created by the RDB. Also, the user should not attempt to edit or modify in any way the coded files used by the database; but copies of these files can be passed from one user to another without any problem, using TOPS GET and REPLACE

commands.

Once referenced, a coded file is active until an explicit command de-activates it. A coded file is not closed at the completion of the database function which referenced it, since there might be other references to it shortly, and excessive numbers of open's and close's would result. Instead, the user must designate the appropriate point to make the file non-active. There are two functions for doing this:

(CLOSECF fname1 fname2 ... fnamen)

closes the coded files fname1 thru fnamen and saves any modifications made to them. Again,

(RETURNCF fname1 fname2 ... fnamen)

also closes fname1 thru fnamen, but does not save any modifications which may have accrued since the files were last opened. If the file has only been referenced by queries, and not modified, then RETURNCF should be used, since it is faster.

Evaluating RETURNCF or CLOSECF with no arguments will return or close all coded files currently active. Both these functions return a list of the coded files closed.

A maximum of thirteen local files are allowed to the database (local untis 2-14), and each coded file uses one of these units if it is just being read, two if written. Thus there is a maximum number of coded files that can be active at once, and the user should be careful to close active files as soon as possible. An error will occur if too many are opened.

BIBLIOGRAPHY

1. Allman, E., Stonebraker, M., and Held, G.
"Embedding a Relational Data Sublanguage in a General Purpose Programming Language", Proceedings of Conference on Data: Abstraction, Definition, and Structure (Salt Lake City, March, 1976) ACM SIGPLAN Notices, Volume 8, No. 2, 1976

2. Bobrow, D.G. and Raphael, B.,
"New Programming Languages for Artificial Intelligence Research", Computing Surveys, Vol. 6, No. 3, September 1974

3. Chamberlin, D.D. and Boyce, R.F.
"SEQUEL - A Structured English Query Language", Proceedings of 1974 ACM-SIGFIDET Workshop of Data Description, Access, and Control, ACM, New York, 1974

4. Chang, C.L.
"DEDUCE - AdDeductive Query Language for Relational Data Bases", in Pattern Recognition and Artificial Intelligence, ed. C.H. Chen, Academic Press, Inc., New York, 1977

5. Codd, E.F.
"A Relational Model of Data for Large Shared Data Banks", CACM, Vol. 13, No. 6, June, 1970

6. Codd, E.F.
"Further Normalization of the Data base Relation Model", in Courant Computer Science Symposia, No. 6, Data Base Systems, New York, May 1971

7. Codd, E.F.
"A Data Base Sublanguage founded on the Relational Calculus", Proceedings of the ACM-SIGFIDET Workshop on Data Access and Control (San Diego, 1971) ACM, New York, 1971

8. Codd, E.F.
"Relational Completeness of Data Base Sublanguages", in Courant Computer Science Symposia, No. 6, Data Base Systems, New York, May 1971

9. Codd, E.F.
"Recent Investigations in Relational Data Base Systems", Information Processing 74, North-Holland Publishing Co., 1974

10. Date, C.J.
An Introduction to Database Systems, Addison-Wesley Publishing Co., 1976.

11. Feldman, J.A. and Rovner, P.D.
"An Algol-based Associative Language", CACM, Vol. 12, No. 6, August 1969

12. Hammer, M.
"Data Abstractions for Data Bases",
Proceedings of Conference on Data: Abstraction, Definition and Structure (Salt Lake City, March 1976) in ACM-SIGPLAN Notices, Vol. 8, No. 2, 1976

13. Hendrix, G.
Partitioned Networks for the Mathematical Modeling of Natural Language Semantics, University of Texas Computer Sciences Dept. Technical Report NL-28, Ph.D. Dissertation, 1975.

14. Konolige, K.
Unpublished report.

15. Konolige, K.
ALISP User's Manual
University of Mass. Computer Center, Amherst, Mass.
August 1975

16. Low, J. and Rovner, P.
Techniques for the Automatic Selection of Data Structures,
University of Rochester Computer Science Dept. Technical Report TR4, 1976

17. McDermott, D.V., and Sussman, J.G.
The CONNIVER Reference Manual
MIT AI Memo 259, May 1972

18. McDermott, D.V.
Very Large PLANNER-type Data Bases,
MIT AI Memo 339, September 1975

19. Reboh, R.
A Preliminary QLISP Manual,
SRI Technical Note 81, August 1973

20. Sussman, J.G., Winograd, T. and Charniak, E.,
MICRO-PLANNER Reference Manual,
MIT AI Memo 203A, December 1971

21. Weyl, S.
An Interlisp Relational Data Base System,
SRI AI Technical Note 116, November 1975