The University of Alberta

CODING SCHEMES FOR THE SEMILATTICE DATA MODEL

by

Kenneth A. Bobey

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring 1987

ISBN 0-315-37724-0

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Kenneth A. Bobey

TITLE OF THESIS: Coding Schemes for the Semilattice Data Model

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1987

(Signed) K A Bobey

Permanent Address:
6004 186 Street
Edmonton, Alberta
T6M 1P3

Date: January 20, 1987

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Coding Schemes for the Semilattice Data Model** submitted by **Kenneth A. Bobey** in partial fulfillment of the requirements for the degree of **Master of Science.**

W. W. Armstrong
Supervisor

Peter Winters

Jia-Huai You

T. A. Marsland

L. Atkinson

Date: Feb 10/1987

ABSTRACT

This thesis describes a coding scheme for the new Semilattice Model of Data. Coding schemes are used internally in the model to represent complex structures in an efficient manner. The proposed coding scheme offers an improvement over other currently known schemes and integrates well with the conceptual level of the model seen by the database designer. To evaluate the new scheme, a comparison is made with two other coding schemes, and a set of experimental measurements taken with the new coding scheme is analyzed.

The language used at the conceptual level of the semilattice model is also presented. This language has the ability to define new types from a set of simple types, other already defined types, and four type constructors. In addition, constraints may optionally be placed on the values of a type. Operators to manipulate objects at the conceptual level are defined and illustrated with examples. Together, type constructors and operators allow complex objects to be defined and manipulated with few restrictions.

This work on the conceptual part of the model, including the new coding scheme, is part of a larger on-going research effort aimed at developing the semilattice model. The model, which is based on the relational data model, has the ability to describe, represent, and manipulate complex objects. Potential applications of the model include such diverse areas as CAD/CAM, VLSI design, Artificial Intelligence, and Computer Graphics.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. The Problem

The Semilattice Model of Data [Arm84] is a new data model that allows complex objects to be represented. The semilattice model is most closely related to the Relational Model of Codd [Cod70] but is considerably extended to the point that even an entire relational database according to Codd's original model can be represented as a single entity.

The two most prominent features of the semilattice model are the conceptual model for describing complex non-normalized data, and the semilattice codes, used internally to refer to common data.

The conceptual model contains four type constructors: set, sequence, tuple, and disjoint union. These constructors can be used with the basic simple types, or with other already defined types, to describe objects.

Semilattice codes are intended to be an efficient and compact way of referring to common data (the intersections in the semilattice). Codes are extremely important in the internal model, not only for reducing storage requirements, but also for performing operations and minimizing disk accesses. This thesis addresses the problem of designing an efficient coding scheme which also possesses the properties desirable for coding schemes in general.

1.2. Original Contribution

The semilattice model is currently being investigated by a group of researchers at the University of Alberta. Most of the initial work on the model is contained in a series of unpublished papers and notes [ArM83, Ger83, Her83]. As a result of these early investigations, the decision was made to implement a prototype semilattice Database Management System (DBMS). For the purposes of the implementation, the work was divided into a conceptual part and a physical part.

The first result of this project was a specification for the language of the conceptual model [Bob84] which has simply been named the Conceptual Language. Next came the work on the physical data structures and their implementation [Sin85] in which a variant of variable length record B-trees, called $C-B^{+}-trees$, was proposed.

This thesis reports on the conceptual work and thus completes the implementation of the prototype semilattice DBMS. A new coding scheme, called structure coding, is presented. Structure coding was selected as the coding scheme for the prototype since it performs better than other coding schemes currently developed. In addition, the Conceptual Language provided for the prototype is described, as several enhancements have been made since the original specification.

1.3. Organization of the Thesis

Chapter two presents all the relevant background material on databases. The chapter is divided into two sections: data models, and the Conceptual Language. The section on data models first briefly reviews the three popular primary data models: network, hierarchical, and relational. Problems and deficiencies of these models are then discussed providing the motivation for a new model. Numerous extensions proposed for the existing models to overcome their deficiencies are then reviewed and evaluated. Finally, the main concepts behind the semilattice model are outlined.

Section two of chapter two presents the Conceptual Language used in the prototype. All examples throughout the thesis use this language. The simple types and four type constructors of the Conceptual Language are described, followed by the various language operations. The chapter concludes with a discussion of how codes have been used in the past and their importance in the semilattice model. A Backus-Naur Form (BNF) specification of the Conceptual Language is contained in Appendix A1.

Chapter three covers the coding schemes for the semilattice model. First, the requirements for a coding scheme are outlined along with some desirable properties of coding schemes. Next, three coding schemes are discussed: absolute, relative, and structure coding. Each scheme is described, illustrated using examples, and then evaluated. Possibilities for other kinds of coding schemes are examined followed by a comparison of all the schemes. The chapter concludes with a discussion of several important practical issues related to the use of codes.

The fourth chapter evaluates structure coding using experimental data and analytical results. First, the prototype implementation is discussed along with its limitations. Storage measurements for the semilattice prototype DBMS and the INGRES DBMS are presented and compared. Next, an informal discussion of performance is given covering several performance topics. The chapter concludes with an examination of codes in distributed database applications.

Chapter five states the conclusions of this thesis based on evaluations of the coding schemes, experimental results, and our experience with coding and the prototype. Directions for further work follow the conclusions.

The source listing for the conceptual part of the prototype implementation is contained in Appendix A2.

Chapter 2

Data Models

Two areas are reviewed in this chapter to establish the basis and motivation for the remainder of the thesis. First, the three primary data models and their deficiencies are outlined. This is followed by a discussion of why a new model is needed along with some proposed models and extensions that have appeared in the literature recently. With the stage thus set, the semilattice model is introduced to show how this model overcomes the problems of the existing models. The second area covered in the chapter is the Conceptual Language of the semilattice model. The language is used in all examples presented here. Types and type constructors are reviewed along with the basic Conceptual Language operations. The concept of codes is also reviewed to serve as an introduction to the main body of the thesis.

2.1. Data Models

Any data model must consist of a notation for expressing data and relationships between data as well as a set of operations for manipulating the data [Ull82]. We often distinguish between the languages corresponding to these two parts of the model, called the data definition language and the data manipulation language respectively. Informal models, such as the entity-relationship model [Che76] are sometimes used to describe data but, strictly speaking, these are not true data models since either the data definition or data manipulation constructs are lacking. In the entity-relationship case, only a method for data definition is provided.

Some definitions of a data model also specifically identify a third component, namely, a set of general integrity rules [Dat83]. Integrity rules and constraints are considered as part of the data definition here.

2.1.1. Primary Models

The three most popular and well known data models in use today are the network, hierarchical, and relational models. These models are reviewed briefly below; a detailed presentation of the models can be found in most database texts [Dat81, Ull82, Mer84]. Since the semilattice model could essentially be described as a generalized version of the non-normalized relational model, the majority of the review below concentrates on the relational case.

The network data model is the result of the work done by the CODASYL Data Base Task Group (DBTG) [COD71, COD78]. A Data Definition Language (DDL) and Data Manipulation Language (DML) are proposed in addition to a Subschema DDL for defining views. Data and relationships are stored in *records* which consist of one or more *fields* or *data items*. Relationships are represented by *links* between records. Although the DDL restricts relationships to be binary many-to-one, arbitrary many-to-many relationships can be represented by introducing *logical records* and additional links. The network model is so named because records and links combine to form a simple directed graph, or network.

Operations supported by the DBTG DML include:

1. FIND, to locate a record,
2. GET, to read a record,
3. INSERT (STORE), to add a new record,
4. DELETE (REMOVE), to delete an existing record, and
5. MODIFY, to change an existing record.

Commercial databases based on the network model include IDMS [Cul78] and TOTAL [Cin78].

The hierarchical data model is similar to the network with the restriction that all

links point from parent to child records. Thus, the records and links in a hierarchical database combine to form a set of trees. Given the tree structure of the model, there are two ways to represent many-to-many relationships: use duplicate records, or introduce additional links. The preferred solution, to eliminate redundancies and the potential for inconsistencies, is to use *virtual records* which are simply pointers to physical records. Bidirectional links are then possible providing the same expressive power as the network model.

Operations in the hierarchical model are similar to the network case. For example, operations to traverse the tree, retrieve a record, insert, delete, and modify records are all provided. One of the most widely used databases today, IBM's IMS [McG77, IBM78], is hierarchical, as is System 2000 [MRI78].

The third and final primary model is the relational data model proposed by Codd [Cod70]. This model is based on the mathematical concept of a relation. Relations are best viewed as unordered tables where the rows correspond to tuples and the columns to attributes. ~~Formally, after~~ After Maier [Mai83], a *relation scheme* R is a finite set of *attribute names*, or simply *attributes*, $\{A_1, A_2, \ldots, A_n\}$. Corresponding to each attribute A_i is a set D_i, $1 \leq i \leq n$, called the *domain* of A_i. A *relation* r on relation scheme R is a finite set of mappings $\{t_1, t_2, \ldots, t_p\}$ from R to $D = D_1 \cup D_2 \cup \cdots \cup D_n$ with the restriction that for each mapping $t \in r$, $t(A_i)$ must be in D_i, $1 \leq i \leq n$. The mappings are called *tuples*. The *active domain* of attribute A_i relative to r is the set of $d \in D_i$ such that ~~there exists $t \in r$~~ with $t(A_i) = d$. A *key* of a relation $r(R)$ is a subset K of R such that for any distinct tuples t_1 and t_2 in r, $t_1(K) \neq t_2(K)$ and no proper subset K' of K shares this property. K is a *superkey* of r if K contains a key of r.

Data manipulation languages for the relational model are classified as either algebraic or predicate calculus languages with the calculus further subdivided into tuple relational and domain relational calculus. Codd introduced relational algebra [Cod70]

and later showed it to be equivalent to the relational calculus [Cod72b]. Algebraic languages apply operations such as union, difference, projection, selection, and join to relations while the calculus specify predicates that tuples must satisfy.

Several normal forms have been developed to eliminate redundancy and prevent inconsistencies for the relational model [Cod72a, Fag77, Ken83a]. We shall be concerned only with the first, second, third, Boyce-Codd, and fourth normal forms, although a variety of other forms exist such as domain-key [Fag81] and fifth [Fag79] normal forms. The normal form definitions below are from Maier [Mai83] where definitions of a functional dependency (FD) and multivalued dependency (MVD) may also be found. Before presenting the definitions, some of Maier's notation must be introduced. The domain of an attribute A is denoted by $dom(A)$. An attribute A is *prime* in R with respect to a set of FDs F if A is contained in some key of R. Otherwise, A is *nonprime* in R. Given a relation scheme R, a subset X of R, an attribute A in R, and a set of FDs F, A is *transitively dependent* upon X in R if there is a subset Y of R with $X \rightarrow Y$, $Y \not\rightarrow X$, and $Y \rightarrow A$ under F and $A \notin XY$. Lastly, an MVD $X \twoheadrightarrow Y$ is *trivial* if for any relation scheme R with $XY \subseteq R$, any relation $r(R)$ satisfies $X \twoheadrightarrow Y$. We are now ready to define the required normal forms. From Maier [Mai83]:

1. A relation scheme R is in *first normal form* (1NF) if the values in $dom(A)$ are atomic for every attribute A in R.
2. A relation scheme R is in *second normal form* (2NF) with respect to a set of FDs F if it is in 1NF and every nonprime attribute is fully dependent on every key of R.
3. A relation scheme R is in *third normal form* (3NF) with respect to a set of FDs F if it is in 1NF and no nonprime attribute in R is transitively dependent upon a key of R.
4. A relation scheme R is in *Boyce-Codd normal form* (BCNF) with respect to a set

of FDs F if it is in 1NF and no attribute in R is transitively dependent upon any key of R.

5. Let F be a set of FDs and MVDs over U. A relation scheme $R \subseteq U$ is in *fourth normal form* (4NF) with respect to F if for every MVD $X \twoheadrightarrow Y$ implied by F that applies to R either the MVD is trivial or X is a superkey for R.

A large research effort has been directed to the studying properties of functional dependencies [Arm74, Ber76, BFH77], multivalued dependencies [BFH77, Lei79], and other dependencies [Ris79, Fag79, Bee80, SaU80, Fag81, Par82, CFP82] in the relational model.

Examples of relational DBMSs include INGRES [SWK76, Sto80], System R [Ast76, Cha81, Bla81], and ORACLE [Sof80].

2.1.2. Deficiencies

All of the three primary models are certainly suitable for a wide variety of applications. However, each model has its deficiencies. Some of these problems are enumerated below keeping in mind that the application often determines the severity of a given problem.

Record and link based data models such as the network and hierarchical suffer two main deficiencies.

1. These models are difficult to use.

   The database designer must fully specify all the links, or relationships, among record types. Usually this involves creating many artificial logical (network) or virtual (hierarchical) records. In addition, the data manipulation languages for these models provide only the most primitive operations (e.g. GET, FIND). Consider the amount of effort required to specify a standard relational algebra operation such as join [Ull82]. As pointed out by Date [Dat81], the relational database

designer must be familiar with one data construct (the relation) while in the DBTG case there are five data constructs: records, DBTG sets, singular sets, ordering, and repeating groups. Besides being unnecessarily complex, the DBTG DDL has been shown to lack some data description concepts required for specialized applications [MMC76].

2. Performance may be poor for complex entities.

Both the network and hierarchical models are well suited to allow efficient implementation. However, when structures with nested many-to-many relationships exist, performance may become a problem because relationships are represented explicitly using pointers. When relationships change, or during updates, each pointer involved must be updated explicitly [Har84]. Some link based systems provide limited solutions to the performance problem. Fastpath is a feature of IMS designed to handle high transaction rates for simple queries [IBM80, Dat81]. This is achieved using a Main Storage Database (MSDB) to reduce access time and limited locking to eliminate waiting time.

Despite the popularity of the relational model, it is not without its own deficiencies.

1. Performance.

Traditionally, performance has been the bane of the relational model since it does not have performance built into the design as the network and hierarchical models do. Although recent improvements in relational products have addressed the performance issue through optimization [Sto83], normalization theory tends to produce less efficient designs because data that could be maintained in a single relation may be decomposed into many relations [Ken83a]. Date points out that a proper implementation of the relational operators requires that the system do optimization to achieve acceptable performance [Dat86].

2. Semantics.

   Several semantical problems have also plagued the relational model. These include: the meaning of data and attributes [Cod79, Sci79], null values [Vas79, Vas80, Lip81], and the assumption of the universal relation [Ken81, Ull83, Ken83b]. Kent enumerates other semantical problems with relationships, naming, and structure in the model [Ken79]. Solutions to most of these problems have been proposed although the suggestions are sometimes less than satisfactory.

A more serious problem yet remains: the inability of the current models to represent complex objects. This problem manifests itself in a number of ways, especially when databases are used for CAD/CAM, VLSI, and graphics applications:

1. Lack of abstraction and data typing capabilities [Hay83, ShW85].
2. Lack of data structuring constructs [JSW83, LeF83, Har84, Spo84].
3. Inflexible methods for representing objects [Gre83, Wil84].
4. Inability to deal with changing, incomplete, and multiple versions of data [HNC84].

Partial solutions to the above problems are possible. For example, structured objects can be represented to some extent with network databases but an inordinate amount of work is required to do so. Schmid reviews some of these issues in integrating complex objects into the conceptual model [Sch77]. None of the primary models offers a complete solution to the complex objects problem.

2.1.3. The Need for a New Model

Researchers have taken two approaches to overcome the deficiencies in the current models. The first approach is to provide extensions to the existing models while the second is to design entirely new models.

2.1.3.1. Proposed Extensions

The following is a brief survey of different extensions proposed for the relational model. First, there are several extensions to solve specific problems. Codd introduced additional semantics [Cod79] while Lipski [Lip81], Vassiliou [Vas79, Vas80], and Keller [KeW85] suggest methods for dealing with null values. Clifford and Warren at Stony Brook introduce the concept of time and show how it can be implemented [ClW83]. Kuck and Sagiv [KuS84] suggest adding network-like links to alleviate some of the performance problems due to decomposition. A variation of the model, called the functional model, has been popularized by several researchers including Buneman and Frankel [BuF79] and Shipman [Shi81].

Other more general extensions have also been proposed to solve many application problems. Haskin and Lorie suggest several extensions to System R [HNC84] such as allowing arbitrarily long tuples, high level operations for manipulating objects, and rudimentary version control. INGRES extensions for hypothetical relations and document processing operators are discussed by Woodfill and Stonebraker [WoS86, Sto86]. Hardwick enhances the model to allow hierarchies and heterogeneous relationships [Har84]. He also proposes an extended relational algebra to handle the new entities. Similarly, Korth extended the model with several features including nested sets and nested tuples [Kor86]. Extensions for non-first-normal-form relations in an office information systems environment are given by Schek and Pistor [ScP82]. Weller and York [WeY84] examine how abstract data types can be represented as relations while Clay-

brook [CCW85] treats views as data abstractions. These last two works form the basis for a method of supporting views and types as meta-data within a relational database.

A significant extension of the relational model is Codd's RM/T [Cod79] which is described in detail by Date [Dat83]. RM/T is based on entities; entities are either characteristic (describe another entity), associative (describe a relationship), or kernel (independently existing entities). Methods for typing, naming, and specifying integrity rules are provided as well as several new operators.

2.1.3.2. New Models

Even with the many proposed extensions, the need for a new model appears to be great. CAD and VLSI design applications require the ability to handle complex engineering data, multiple versions, and a variety of integrity constraints [HNC84, BaK85]. Graphics applications require the ability to handle compound, complex objects using a high level data model [Gre83, Spo84]. Even statistical and scientific applications require complex data types, a semantically rich model, and temporal and meta-data with a supporting user interface [ShW85].

Many new data models have been introduced into this environment. Lee and Fu at Purdue University adopted Constructive Solid Geometry (CSG) as the basis for a model [LeF83] and augmented the relational query language SEQUEL to support their new structures. Kitagawa and his group [Kit84] described a model called Formgraphics to integrate text, diagrams, and images into a graphics database. Ulfsby, Meen, and Oian introduced Tornado [UMO82], based on the network model, for solid modeling and CAD applications. The PICCOLO database system was developed at the University of Tokyo by Yamaguchi and Kunii for handling pictorial data such as LANDSAT images [YaK82]. Many models have been introduced for specialized problems including those by McLellan [McL85] for VLSI design, Schell and Mercer [ScM85] for CAD, and

Su [Su86] for Computer Integrated Manufacturing (CIM). Sockut, at the U.S. National Bureau of Standards, suggested a framework able to handle logical level changes in databases [Soc85]. Examples of such changes include mapping between models, restructuring schemas, and revising database programs.

In contrast to these application oriented approaches, several models having a more mathematical foundation have been introduced. Jacobs describes a database logic [Jac82] which generalizes the principles of the three primary models. Hammer and McLeod introduced the Semantic Database Model (SDM) containing high level primitives for structuring and modeling data [HaM81]. Kuper and Vardi [KuV85] designed the Logical Data Model (LDM) which is based on basic types, references, composition, collection, and classification. The Format Model of Hull and Yap [HuY84] is similarly based on the constructs of collection, composition, and classification. A database programming language called Galileo supporting these semantic features in addition to abstraction (types and modularization) has been proposed by Albano, Cardelli, and Orsini [ACO85].

Many of the extensions and new models are designed for specific problems or applications and do not constitute truly general, unified solutions. Also, with the exception of more recent theoretical work, many of the proposed models are not based on the solid mathematical foundation that the relational model is.

2.1.4. Semilattice Model

The semilattice data model [Arm84] is based on the relational model and the relational algebra of Codd [Cod70]. One of the major differences between the models is that the semilattice model does not impose the requirement of normalization although the database designers may still normalize relations if they so wish.

Two important features of the semilattice model are the Conceptual Language

and the semilattice codes. The Conceptual Language is a high level interface that permits types, abstractions, and complex operations. It addresses all of the deficiencies cited previously such as abstraction capability, data structuring constructs, flexibility in dealing with objects, definition and manipulation of compound and complex objects, temporal and meta-data, and multiple versions. The semilattice codes make many of the Conceptual Language features possible. These codes permit implementation of complex structures, many-to-many relationships, and allow efficient implementation, thereby addressing performance concerns.

2.2. Semilattice Conceptual Language

The Conceptual Language of the semilattice model was first specified as part of a larger technical report [Bob84]. The language presented here includes some enhancements to the original specification and represents the version currently implemented in the prototype. A BNF grammer for the language is given in Appendix A1.

The language consists of two parts: constructs for data definition and operations for data manipulation. Constructs and operations are well integrated and there is rarely any reason to separate the two other than for discussion purposes.

2.2.1. Data Definition

A semilattice *type* consists of a *domain* (set of values) and a set of operations. Operations are defined in a following section. The Conceptual Language allows named types to be defined using type expressions. A *type expression* is either a simple type or a complex type; complex types are formed by applying type constructors to other already defined types.

Simple types are predefined in the semilattice model and include integer, real, boolean, and string with the usual Pascal values. For example,

- 3 is an integer,
- 3.1415927 is a real,
- true is a boolean, and
- "Smith" is a string.

Four *type constructors*: set, sequence, tuple, and union are used to define complex types.

1. Set scheme $\{T\}$.

   The domain of $\{T\}$ is the set of all sets of values in the domain of type T. A value of the set type is of the form $\{e_1, e_2, \ldots, e_n\}$ where each e_i is a distinct element of type T.

2. Sequence scheme $<T>$.

   The domain of $<T>$ is the set of all finite sequences of values in the domain of type T. A value of the sequence type is of the form $<e_1, e_2, \ldots, e_n>$ where the elements e_i form an array indexed by i and need not be distinct.

3. Tuple scheme $(A_1:T_1, A_2:T_2, \ldots, A_n:T_n)$.

   The domain of a tuple scheme is the set of all tuples with some value from each domain of type T_i assigned to the corresponding distinct attribute A_i. This is a tuple in the relational sense where a set of such tuples is called a relation. A value of the tuple type is of the form $(A_1 = v_1, A_2 = v_2, \ldots, A_n = v_n)$ or, using the order of the attributes in the tuple scheme definition, $(v_1, v_2, \ldots, v_n)$.

4. Union scheme $(t_1:T_1 \mid t_2:T_2 \mid \ldots \mid t_n:T_n)$.

   The domain of a union scheme is the union of the sets of all tagged values in each domain of type T_i. Each tag t_i must be distinct and for this reason the scheme is also referred to as a disjoint union. Union schemes are equivalent to Pascal variant records. A value of the union type is of the form $(\text{tag} = t_i, \text{value} = v_i)$ or, more concisely, (t_i, v_i).

Once a type has been defined and named, that type may be used in any type expression, without restriction, to define new types. This allows arbitrarily complex objects to be defined. For example, if we define a tuple scheme to represent a point in two dimensional space

point = (x : real , y : real)

a polygon can then be defined as a set of points and a circle as a center point and a radius.

polygon = { point }
circle = (center : point , radius : real)

Next, a shape can be defined as a polygon or circle together with some identification.

shape = (name : string , fig : (poly : polygon | circ : circle))

Examples of values of the shape tuple scheme are as follows:

("triangle" , (poly , { (1.1,2.1) , (3.1,2.5) , (4.9,1.7) }))
("unit circle" , (circ , ((0.0,0.0) , 1.0)))

Note that a tuple scheme combined with a set scheme represents the traditional relation. Union schemes are used in the definition of heterogeneous structures as shown above. A special tag, called the *null tag*, is predefined in the union scheme. This tag allows the database schema to be changed without affecting the underlying data or programs which use that data. To illustrate, suppose the shape tuple scheme was originally defined to include only polygons.

shape = (name : string , fig : polygon)

Now, if a circle is added as a new shape, the new definition of the scheme becomes:

shape = (name : string , fig : (null : polygon | circ : circle))

The Conceptual Language also permits *constraints* to be placed on the domain and values assigned to types. This is accomplished when a type is first defined using the

following syntax:

Type_name = Type_expression, constraint: "Qualification"

where Qualifications are discussed in the section on operations. An English *meaning* clause can also be appended to a type declaration for documentation purposes. As an example, to restrict the point tuple scheme to contain only positive real values,

```
point = ( x : real , y : real ),
        constraint: "x > 0 and y > 0",
        meaning: "point must lie in the first quadrant"
```

Constraints will not be discussed further here other than noting the Conceptual Language provides a method for their declaration.

2.2.2. Example Database

Throughout the remainder of the thesis a sample database will be used for all examples. The database is taken from the Happy Valley Food Coop (HVFC) example in Ullman [Ull82] but is suitably modified to illustrate some of the semilattice features. Four tuple schemes are defined in the example.

```
Member = ( name : string , address : string , balance : real )
Priced_item = ( item : string , price : real )
Order = ( num : integer , purchaser : Member , item : Priced_item ,
          quantity : integer )
Supplier = ( name : string , address : string , stock : { Priced_item } )
```

The variables representing the four relations in the database are as follows.

```
members : { Member }
goods : { Priced_item }
orders : { Order }
suppliers : { Supplier }
```

An instance of the HVFC database is shown below.

members		
name	address	balance
"Brooks, B."	"7 Apple Rd."	+10.50
"Field, W."	"43 Cherry La."	0.00
"Robin, R."	"12 Heather St."	-123.45
"Hart, W."	"65 Lark Rd."	-43.00

goods	
item	price
"Granola"	1.29
"Lettuce"	0.89
"Sunflower Seeds"	1.09
"Whey"	0.70
"Curds"	0.80
"Granola"	1.25
"Unbleached Flour"	0.65
"Lettuce"	0.79
"Whey"	0.79
"Sunflower Seeds"	1.19

orders			
num	purchaser	item	quantity
1	("Brooks, B.","7 Apple Rd.",+10.50)	("Granola",1.29)	5
2	("Brooks, B.","7 Apple Rd.",+10.50)	("Unbleached Flour",0.65)	10
3	("Robin, R.","12 Heather St.",-123.45)	("Granola",1.25)	3
4	("Hart, W.","65 Lark Rd.",-43.00)	("Whey",0.70)	5
5	("Robin, R.","12 Heather St.",-123.45)	("Sunflower Seeds",1.09)	2
6	("Robin, R.","12 Heather St.",-123.45)	("Lettuce",0.79)	8
7	("Hart, W.","65 Lark Rd.",-43.00)	("Whey",0.70)	3
8	("Brooks, B.","7 Apple Rd.",+10.50)	("Granola",1.29)	2

suppliers		
name	address	stock
"Sunshine Produce"	"10 River St."	{("Granola",1.29),("Lettuce",0.89), ("Sunflower Seeds",1.09)}
"Purity Foodstuffs"	"180 Industrial Rd."	{("Granola",1.25),("Curds",0.80), ("Whey",0.70),("Unbleached Flour",0.65)}
"Tasti Supply Co."	"17 River St."	{("Lettuce",0.79),("Whey",0.79), ("Sunflower Seeds",1.19)}
"Acme Supply"	"20 Industrial Rd."	{("Granola",1.25),("Curds",0.80), ("Whey",0.70),("Unbleached Flour",0.65)}

Figure 2.1 HVFC Example Database Instance

Note that relations are not physically stored in the semilattice database as shown

above. However, the database instance appears this way externally.

2.2.3. Data Manipulation

There are three classes of operations in the Conceptual Language: arithmetic and conditional operations, algebraic operations, and update operations. Arithmetic operators (+, -, *, /) are defined only on integer and real types. Conditional operators for ordering (<, >, <=, >=) are defined on integer, real, and string types. Equality and inequality (=, !=) are defined on all types. A subset operator (in) is also defined for sets. All arithmetic and conditional operators have the usual Pascal semantics.

The Conceptual Language algebraic operations are defined below. All definitions are similar to those of relational algebra except for the extension to arbitrary types. In each definition below the target variable V must be of the same type as the operation result. V_1 and V_2 are of the set type and may be either declared variables or explicit values.

1. Union: V := union(V_1, V_2).

   The union V of sets V_1 and V_2 is the set of all elements which belong to at least one of V_1 or V_2.

2. Difference: V := difference(V_1 , V_2).

   The difference V of sets V_1 and V_2 is the set of all elements which belong to V_1 but not V_2.

3. Product: V := product(V_1 , V_2).

   The cartesian product V of sets V_1 and V_2 is the set of tuples whose first element is in V_1 and whose second element is in V_2. When V_1 and V_2 are relations where V_1 has arity v_1 and V_2 has arity v_2, the product result V is a relation of arity $v_1 + v_2$. For relations, product is normally used in conjunction with a selection

to perform a join.

4. Projection: $V := V_1$.

Projection is defined as in the relational sense on a set of tuples. The common attributes in V_1 are projected onto V. Attributes may also be reordered by the projection.

5. Selection: $V := V_1$ where Q.

Selection is also defined as in the relational sense on a set of tuples. Only those tuples in V_1 satisfying the qualification formula Q are assigned to V. The syntax of the Qualification is given in Appendix A1.

To illustrate how these operations are used, consider adding a new member to the members relation. One way to accomplish this is using union.

members := union(members , {("Doe, J." , "30 Apple Rd." , 0.00)})

The members relation in figure 2.2 shows the result after taking the union.

members		
name	address	balance
"Brooks, B."	"7 Apple Rd."	+10.50
"Field, W."	"43 Cherry La."	0.00
"Robin, R."	"12 Heather St."	-123.45
"Hart, W."	"65 Lark Rd."	-43.00
"Doe, J."	"30 Apple Rd."	0.00

Figure 2.2 Union Example

Next, suppose we wish to obtain a projection of the orders relations showing only the quantity of each item on order. A new tuple scheme must be defined and corresponding relation declared

```
on_order = ( quantity : integer , item : Priced_item )
ordered : { on_order }
```

before taking the projection:

```
ordered := orders
```

Figure 2.3 shows the new projected relation.

ordered	
quantity	item
5	("Granola",1.29)
10	("Unbleached Flour",0.65)
3	("Granola",1.25)
5	("Whey",0.70)
2	("Sunflower Seeds",1.09)
8	("Lettuce",0.79)

Figure 2.3 Projection Example

Finally, consider a selection operation on the goods relation of items priced higher than one dollar.

```
expensive_goods : { Priced_item }
expensive_goods := goods where goods.price > 1.00
```

The resulting relation is shown below.

goods	
item	price
"Granola"	1.29
"Sunflower Seeds"	1.09
"Granola"	1.25
"Sunflower Seeds"	1.19

Figure 2.4 Selection Example

Other algebraic relations which are not part of the Conceptual Language can be

obtained using the five basic operations. Intersection is computed by taking differences; natural join by taking a product followed by a projection; and generalized θ-join by taking a product followed by a selection for a θ relationship.

Three update operations are also provided in the Conceptual Language for convenience when updating relations.

1. Insert: insert V_1 into V_2.

   Insert is a special case of union and is equivalent to V_2 := union(V_2 , V_1).

2. Delete: delete V_1 from V_2.

   Delete is a special case of difference equivalent to V_2 := difference(V_2 , V_1).

3. Modify: modify V_1 with M where Q.

   Modify is equivalent to a delete followed by an insert where the modified tuples are inserted. The operation is quite powerful since a selection can be specified (Qualification Q) to restrict the target of the modification to selected tuples.

As an example, consider the modify operation with a qualification. Suppose 5% interest is to be charged to all members in figure 2.1 with unpaid balances. This operation is given by:

modify members with members.balance*1.05 where members.balance < 0.0

Figure 2.5 shows the modified members relation.

members		
name	address	balance
"Brooks, B."	"7 Apple Rd."	+10.50
"Field, W."	"43 Cherry La."	0.00
"Robin, R."	"12 Heather St."	-129.62
"Hart, W."	"65 Lark Rd."	-45.15

Figure 2.5 Modify Example

2.3. Codes

We are now ready to begin the main discussion of this thesis: codes and coding schemes. A *code* is an abbreviated symbol used to refer to an entity. Typically, codes are small positive integers. Note that codes are not links in the network sense so that link based relational implementations, such as those proposed by Kuck and Sagiv [KuS82, KuS84], do not constitute coding schemes. Codes have been used in other diverse database applications such as partial match retrieval [SaR83].

In the relational model codes are often called *tuple identifiers* (TIDs), or *surrogates* as introduced in Hall [HOT76] and Codd [Cod79]. The difference in terminology is primarily due to the fact that a surrogate, once assigned, is guaranteed never to change [Dat83]. Several relational models actually use tuple identifiers as part of the underlying implementation. For example, figure 2.6 from Date [Dat81] illustrates the two-part TIDs used in System R. The first part of the TID contains the page number of the page containing the tuple. The second part contains an offset from the bottom of the page to a second offset which locates the tuple within the page.

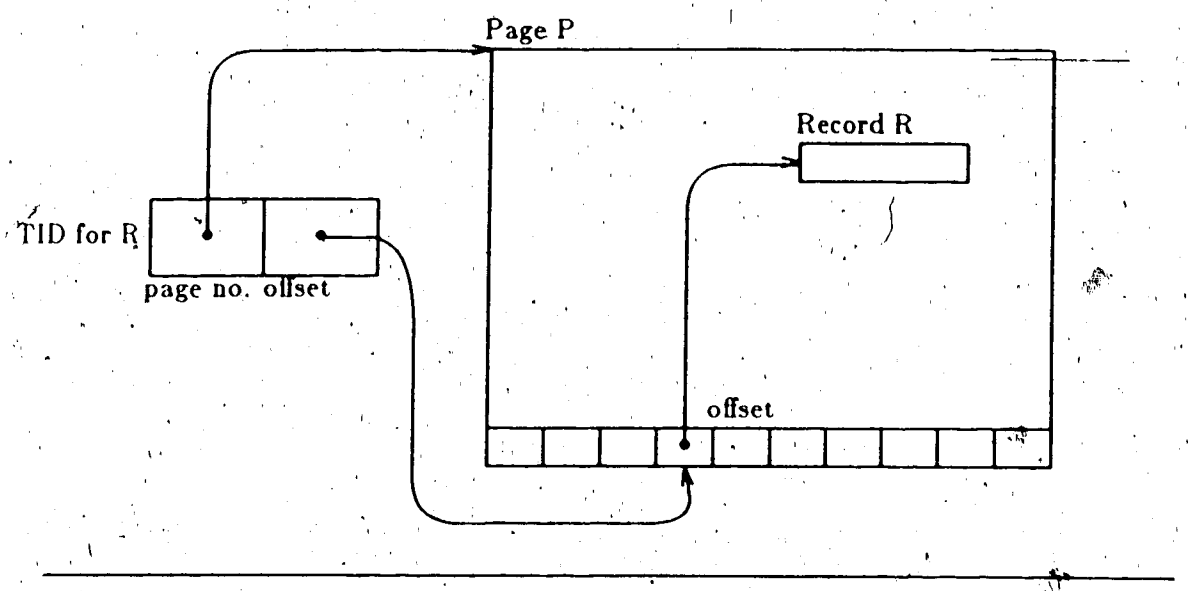

Figure 2.6 System R Implementation of TIDs

TIDs also permit tuples to be accessed directly in System R thus improving performance.

A more interesting application of codes and tuple identifiers is in the representation of complex structures. One method is to represent objects as abstract types (ADTs) [SRG83]. In general, techniques for classifying, encoding, and decoding data types have been well known for some time [Wie77]. However, this method requires adding specialized functions to the database for each new ADT. In addition, changes are required to the database secondary indexing to allow accesses based on the new types.

A second approach which overcomes these problems is to use tuple identifiers to link together related objects [Lor82]. Each tuple in every relation is assigned a tuple identifier. Several link relations containing only TIDs are then used to describe relationships. Specialized codes, called *identifier values* and *reference types* [LoP83] may also be used as part of the methodology for implementing complex objects. Spooner [Spo84] gives an extended example of a graphics application using the second approach

to represent complex data with a relational model. Unfortunately, the example illustrates one of the drawbacks of the method as Spooner observes:

> At first glance this appears more complicated than necessary, but it must be recognized that most of the complexity arises because QUEL [the query language] is not equipped to handle complex objects directly.

Spooner finds it necessary to modify QUEL to make the complexity more manageable.

Therefore, codes, or tuple identifiers, are of considerable utility in the relational model since they allow representation of complex objects and aid performance. Some problems remain however, such as integrating the flexibility provided by the codes into the model. With the stage thus set, we are now prepared to examine the issue of codes in the semilattice model.

Chapter 3

Semilattice Coding Schemes

Chapter three explores the concepts and applications of coding schemes in the semilattice model. First, the requirements and properties for a coding scheme are stated. Two existing schemes, absolute and relative coding, are then discussed. A proposed new coding scheme, called structure coding, is presented next. All three schemes along with some variations are then compared and the main results examined. The chapter concludes with a discussion of several issues related to the use of codes in practice.

3.1. Requirements for a Coding Scheme

Besides the high level Conceptual Language, the second significant feature of the semilattice model is the internal use of codes. Any coding scheme can be used by the model provided the scheme meets several criteria. Those criteria that are considered necessary, that is, must be met, are stated as requirements. The remaining criteria which are desirable, that is, should be met for performance or other reasons, are stated as desirable properties.

Requirements which a semilattice coding scheme must meet are as follows.

Provide a method to refer to common data. (R1)

This requirement results from the fundamental concept in the semilattice model called the *semilattice condition* defined by Armstrong [Arm84]:

> A directed graph can be formed from tuple scheme names or relation names by using arcs corresponding to tuple scheme mappings, whereby the tuple scheme at the upper end of the arc will, by convention, define an object which is part of the object corresponding to the lower end.
>
> The semilattice condition for forming a diagram is the following: if two relation schemes have a common part (i.e. relations which are equal according to the semantics of the database, up to renaming of attributes and domains), then the scheme for the common part must appear in the diagram.

Figure 3.1 illustrates the semilattice condition. Two relations, R_1 and R_2, have a non-empty intersection R_3. The intersection R_3 is stored only once and codes are used to refer to it [Arm84].

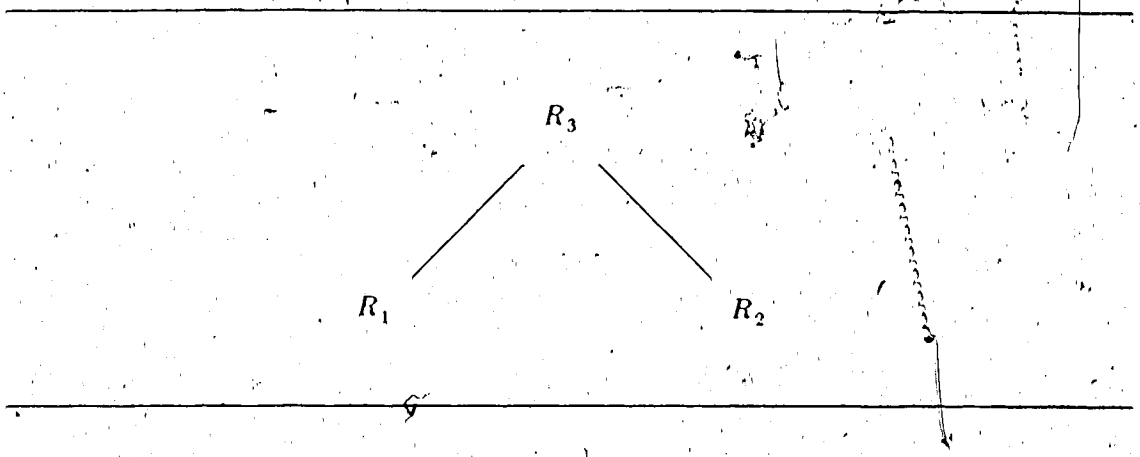

Figure 3.1 Semilattice Condition

Like normalization in the relational model, the use of codes avoids repeating data, thus reducing storage requirements, and helps prevent inconsistencies in the database.

A second requirement of coding schemes is:

Provide a compact method of reference. (R2)

There are several reasons why a coding scheme should be as compact as possible. Compact codes reduce memory and storage requirements in addition to increasing the amount of information that can be transferred in a single disk block access. When the distribution of data elements is known a priori the optimum coding procedure is Huffman coding [Sam76]. Since distributions are rarely known a priori in dynamic database applications the next best coding procedure is to map the data elements in a one-to-one correspondence with the set of positive integers.

Desirable properties of coding schemes are as follows.

Provide a code which is unaffected by changes to the underlying data object. (P1)

Property (P1) essentially states that a code should be a surrogate in the usual sense [Dat83]. That is, the code is guaranteed never to change and is permanently assigned to the object. The characteristics of the object may change but the object being referred to, and thus the code, does not change. The reason (P1) is desirable is to avoid reorganizations when entities change. Changing the value of an entity and then changing its code means all coded references to the entity must also be updated. Such references may be scattered throughout the database and updating these could be very time consuming in a large system.

There is a second desirable property of coding schemes:

Provide a single unique code for each entity. (P2)

Property (P2) has implications for how certain operations are carried out. When codes have this uniqueness property then two entities are equal if and only if their codes are equal. This means codes can be used directly for several operations in place of the actual entity. Union, difference, and selection for equality and inequality can all operate on codes. However, a generalized selection operation on codes requires that the codes be range preserving. Checking for duplicate entities following operations can also be done by testing for duplicate codes. Using codes in operations is extremely efficient, especially for complex objects.

The third and final desirable property of coding schemes is:

Provide a method to eliminate repeated values in representing objects. (P3)

Property (P3) is actually a property of the coding methodology rather than than the code itself. Although coding schemes remove most redundancy in the database, there are cases when repeated data remains. The exception occurs when complex objects are

represented implicitly as part of other entities instead of explicitly as declared variables. The examples in the following sections illustrate this kind of redundancy.

What is typically coded in a semilattice database? Two heuristic rules are used to decide codability.

1. Values which are already dense need not be coded [Arm86].
2. Non-dense simple values which are unlikely to repeat need not be coded.

Using these two rules with the Conceptual Language means integers and booleans are not coded since they are already dense. Interestingly, this also implies that integer codes need not be recoded. Reals and strings are also not coded since they are unlikely to repeat. The exception for both reals and strings is when only a very small subset of the domain will be used and thus repetition will occur. Using strings as an example, a suitably small domain would be the names of Canadian cities with a population greater than 500,000. This small set of names can be coded using four bits.

Values of types formed by type constructors may or may not be coded depending on the coding scheme. Sets, sequences, and union values (including tags) can all be coded. Tuples are coded in each of the coding schemes presented next.

3.2. Absolute Coding

Absolute coding is the simplest and most straightforward coding scheme for the semilattice model. Under the scheme, a unique identifier is assigned to each value in a set. Each instance of the value in the other entities in the database is replaced by the assigned identifier or code. An entity which is a set or sequence of tuples is thus represented by a set or sequence of codes respectively.

Although there are no restrictions on how codes are selected for absolute coding, monotonically increasing positive integers are typically used. This makes absolute codes quite similar to tuple identifiers in the relational model. Figure 3.2 shows how

the HVFC example database is represented internally using absolute coding. The notation *e/* means the code for entity e.

members			
name	address	balance	members/
"Brooks, B."	"7 Apple Rd."	+10.50	0
"Field, W."	"43 Cherry La."	0.00	1
"Robin, R."	"12 Heather St."	-123.45	2
"Hart, W."	"65 Lark Rd."	-43.00	3

goods		
item	price	goods/
"Granola"	1.29	0
"Lettuce"	0.89	1
"Sunflower Seeds"	1.09	2
"Whey"	0.70	3
"Curds"	0.80	4
"Granola"	1.25	5
"Unbleached Flour"	0.65	6
"Lettuce"	0.79	7
"Whey"	0.79	8
"Sunflower Seeds"	1.19	9

orders				
num	purchaser	item	quantity	orders/
1	0	0	5	0
2	0	6	10	1
3	2	5	3	2
4	3	3	5	3
5	2	2	2	4
6	2	7	8	5
7	3	3	3	6
8	0	0	2	7

suppliers			
name	address	stock	suppliers/
"Sunshine Produce"	"16 River St."	{0,1,2}	0
"Purity Foodstuffs"	"180 Industrial Rd."	{5,4,3,6}	1
"Tasti Supply Co."	"17 River St."	{7,8,9}	2
"Acme Supply"	"20 Industrial Rd."	{5,4,3,6}	3

Figure 3.2 Absolute Coding Internal Representation

Now let us evaluate absolute coding using the requirements and properties for coding schemes stated earlier. Both requirements are satisfied since absolute coding provides a method to refer to common data (R1) and provides a compact method of reference (R2). Also note that when a mapping to positive integers is used, as in the example, the resulting codes are the most efficient possible as discussed in the previous section.

Two properties are also satisfied by the scheme. Absolute codes are not affected by changes to the underlying data objects (P1) and the codes are unique (P2) since a one-to-one mapping to the positive integers is used. To illustrate these properties, when the value of any attribute in figure 3.1 is changed, the corresponding code remains unchanged. An interesting case occurs when tuples are selectively modified such that duplicate tuples result. If the database permits duplicates, it is possible that (P2) will be violated since modifications can lead to different codes assigned for the same tuple value.

Unfortunately, absolute coding does not satisfy property (P3) since the scheme fails to remove repeated values. The suppliers relation in the example contains redundancy in the stock attribute (bold). A complex structure, namely, a set of priced items, is stored twice. In the example this redundancy is not too serious but in a real application a supplier could easily stock several hundred or thousand items and affiliated or subsidiary companies could also stock those same items. Such a situation would lead to large amounts of redundancy.

3.3. Relative Coding

A second coding scheme for the semilattice model is relative coding. As with absolute coding, relative coding assigns a unique identifier to each value in a set; other instances of that value are then replaced by codes. The difference between the two schemes is the method used to generate codes and the length of the resulting codes.

The basic concept behind relative coding is to assign codes to values in a set of tuples which have the same value for a given fixed set of attributes. The relative code and uncoded attributes are combined to form a complete code for the tuple. Given these guidelines, relative codes can be formed in several ways. The method used here is to concatenate the relative code first followed by the uncoded attributes in the order they appear in the tuple scheme.

Figure 3.3 illustrates the HVFC internal representation using relative coding. Note that when all attributes in a tuple scheme are coded, as with the members, goods, and suppliers relations, relative coding simply degenerates to absolute coding. The notation c_i/a_j means the code for attributes c_i relative to attributes a_j.

---

members			
name	address	balance	nab/
"Brooks, B."	"7 Apple Rd."	+10.50	0
"Field, W."	"43 Cherry La."	0.00	1
"Robin, R."	"12 Heather St."	-123.45	2
"Hart, W."	"65 Lark Rd."	-43.00	3

goods		
item	price	ip/
"Granola"	1.29	0
"Lettuce"	0.89	1
"Sunflower Seeds"	1.09	2
"Whey"	0.70	3
"Curds"	0.80	4
"Granola"	1.25	5
"Unbleached Flour"	0.65	6
"Lettuce"	0.79	7
"Whey"	0.79	8
"Sunflower Seeds"	1.19	9

orders				
num	purchaser	item	quantity	nq/pi
1	0	0	5	0:0:0
2	0	6	10	0:0:6
3	2	5	3	0:2:5
4	3	3	5	0:3:3
5	2	2	2	0:2:2
6	2	7	8	0:2:7
7	3	3	3	1:3:3
8	0	0	2	1:0:0

suppliers			
name	address	stock	nas/
"Sunshine Produce"	"16 River St."	{0,1,2}	0
"Purity Foodstuffs"	"180 Industrial Rd."	**{5,4,3,6}**	1
"Tasti Supply Co."	"17 River St."	{7,8,9}	2
"Acme Supply"	"20 Industrial Rd."	**{5,4,3,6}**	3

Figure 3.3 Relative Coding Internal Representation

Relative coding satisfies the two requirements of coding schemes. Both a method to refer to common data (R1) and a compact method of reference (R2) are provided. However, a significant problem with relative coding is that the length of codes depends on the number of uncoded attributes. Furthermore, only integers, booleans, and other codes can be left uncoded otherwise non-integer values (reals, strings, and complex types) are introduced into the code.

Relative coding only satisfies one of the three desirable properties of codes, that is, unique codes are provided (P2). The duplicate tuple problem may result in non-

unique codes as discussed earlier. Property (P1) is not satisfied since changing the value of an underlying data object will always change the code when uncoded attributes are modified. To illustrate, consider the first order in the orders relation, (1,0,0,5), which is assigned "0:0:0" as a code. If the item on order is in error and is subsequently changed to Unbleached Flour, the code for which is "6", the new tuple becomes (1,0,6,5) and the corresponding code "0:0:6". Not only must all references to this tuple throughout the database be changed, but now a worse situation has developed: a single code "0:0:6" refers to two completely different tuples. Recoding the relative part for one of the tuples is required to restore the uniqueness property (P2).

Repeated values are not removed by relative coding and thus property (P3) is violated. The bold part of the suppliers relation again identifies the redundancy. Relative coding does have one advantage over absolute coding however. Generalized selection using codes is supported for uncoded attributes. This is due to the fact that relative codes are range preserving for uncoded attributes since the code contains additional information, in this case, the actual value.

3.4. Structure Coding

Structure coding is a new coding scheme proposed here for the semilattice model to overcome the deficiencies of the previous methods. Structure coding is considerably different in principle from absolute and relative coding although the codes generated are often similar. The previous schemes code tuples of relations while structure coding deals with **structures**. A *structure* is defined as a tuple, set, or sequence scheme. As with absolute coding, monotonically increasing positive integers are assigned for codes.

The method used by structure coding is as follows.

1. Code all values in the active domain of each structure type. All attributes, tagged values, and declared variables having the same structure type are coded together.

2. Replace all references to tuples, sets, and sequences with the codes generated in step (1).

It may be necessary in step (1) to create additional internal relations to code structures which do not have an explicitly declared variable of that type. This is best illustrated with an example. Figure 3.4 shows the internal representation for the HVFC database using structure coding. First, notice that the stock attribute of the Supplier structure now contains codes rather than Priced_item sets. These codes refer to the internal relation called {Priced_item} containing sets of priced items. Also, note that it is structures (Member, Priced_item, Order, Supplier, and {Priced_item}) which are coded as opposed to variables (members, goods, orders, suppliers) in the previous schemes.

Member			
name	address	balance	Member/
"Brooks, B."	"7 Apple Rd."	+10.50	0
"Field, W."	"43 Cherry La."	0.00	1
"Robin, R."	"12 Heather St."	-123.45	2
"Hart, W."	"65 Lark Rd."	-43.00	3

Priced_item		
item	price	Priced_item/
"Granola"	1.29	0
"Lettuce"	0.89	1
"Sunflower Seeds"	1.09	2
"Whey"	0.70	3
"Curds"	0.80	4
"Granola"	1.25	5
"Unbleached Flour"	0.65	6
"Lettuce"	0.79	7
"Whey"	0.79	8
"Sunflower Seeds"	1.19	9

Order				
num	purchaser	item	quantity	Order/
1	0	0	5	0
2	0	6	10	1
3	2	5	3	2
4	3	3	5	3
5	2	2	2	4
6	2	7	8	5
7	3	3	3	6
8	0	0	2	7

{Priced_item}	
{Priced_item}	{Priced_item}/
{0,1,2}	0
{5,4,3,6}	1
{7,8,9}	2

Supplier			
name	address	stock	Supplier/
"Sunshine Produce"	"16 River St."	0	0
"Purity Foodstuffs"	"180 Industrial Rd."	1	1
"Tasti Supply Co."	"17 River St."	2	2
"Acme Supply"	"20 Industrial Rd."	1	3

Figure 3.4 Structure Coding Internal Representation

Structure coding satisfies both requirements of coding schemes: a method to refer to common data (R1) and a compact method of reference (R2) are provided. The one-to-one mapping to positive integers ensures the codes are the most efficient possible. Integer codes also mean properties (P1) and (P2) are satisfied as with absolute coding. Duplicate structures, if permitted, remain a problem that can lead to (P2) violations.

Unlike absolute and relative coding, structure coding provides a method of eliminating repeated values thus satisfying property (P3). This can be seen in figure 3.4 where no redundancy is present in any structure. The redundancy in the stock attribute of the Supplier structure seen in previous examples has been corrected.

Structure coding combined with the Conceptual Language gives the database designer control over how coding will be carried out. Thus, the designer ultimately

decides how efficient the database will be. However, efficiency is often determined by the semantics and assumptions of the data definition.

To illustrate these points, consider the HVFC example. The definition of Member and Order implies that only members may place orders. If a new order is placed by an individual who is not a member then two actions are possible. The database can either

- reject the order since the individual is not a member, or
- accept the order and automatically add the new member.

Figure 3.5 shows the Member and Order structures following a new order if the second approach is taken. A new member has been inserted in the Member structure and a new order placed, all with a single operation.

Member

name	address	balance	Member/
"Brooks, B."	"7 Apple Rd."	+10.50	0
"Field, W."	"43 Cherry La."	0.00	1
"Robin, R."	"12 Heather St."	-123.45	2
"Hart, W."	"65 Lark Rd."	-43.00	3
"Smith, A."	"20 Apple Rd."	-2.50	4

Order

num	purchaser	item	quantity	Order/
1	0	0	5	0
2	0	6	10	1
3	2	5	3	2
4	3	3	5	3
5	2	2	2	4
6	2	7	8	5
7	3	3	3	6
8	0	0	2	7
9	4	5	2	8

Figure 3.5 Structure Coding Example (I)

However, suppose orders are also accepted from non-members but a higher price is charged. This means the purchaser in the Order structure is semantically different

from the Member structure and the two structures should not be coded together. The database designer defines new structures as follows.

```
Member = ( name : string , address : string , balance : real )
Person = ( name : string , address : string , balance : real )
Order = ( num : integer , purchaser : Person , item : Priced_item ,
          quantity : integer )

members : { Member }
orders : { Order }
```

Figure 3.6 shows how the new structures are represented using the same data as in figure 3.5. An internal relation called {Person} has been created containing both members and non-members.

Member			
name	address	balance	Member/
"Brooks, B."	"7 Apple Rd."	+10.50	0
"Field, W."	"43 Cherry La."	0.00	1
"Robin, R."	"12 Heather St."	-123.45	2
"Hart, W."	"65 Lark Rd."	-43.00	3

{Person}			
name	address	balance	{Person}/
"Brooks, B."	"7 Apple Rd."	+10.50	0
"Field, W."	"43 Cherry La."	0.00	1
"Robin, R."	"12 Heather St."	-123.45	2
"Hart, W."	"65 Lark Rd."	-43.00	3
"Smith, A."	"20 Apple Rd."	-2.50	4

Order				
num	purchaser	item	quantity	Order/
1	0	0	5	0
2	0	6	10	1
3	2	5	3	2
4	3	3	5	3
5	2	2	2	4
6	2	7	8	5
7	3	3	3	6
8	0	0	2	7
9	4	5	2	8

Figure 3.6 Structure Coding Example (II)

The database instance above is less efficient than that in figure 3.5 due to the repeated member data. However, the Conceptual Language provides the database designer with a constructor to overcome this problem: disjoint union. By using disjoint union, members and non-members can be semantically distinguished and the database remains efficient. The database definition using disjoint union is shown below. Note that the two parts of the union must be different structures to force separate coding.

```
Member = ( name : string , address : string , balance : real )
Person = ( name : string , address : string , balance : real )
Order = ( num : integer , purchaser : ( mem : Member | non : Person ) ,
          item : Priced_item , quantity : integer )

members : { Member }
orders : { Order }
```

Figure 3.7 shows the representation of the new structure. The purchaser attribute of the Order structure now contains disjoint union values in which the first element is the tag identifying either the Member (0) or Person (1) structure and the second element is the code with respect to the tagged structure.

Member			
name	address	balance	Member/
"Brooks, B."	"7 Apple Rd."	+10.50	0
"Field, W."	"43 Cherry La."	0.00	1
"Robin, R."	"12 Heather St."	-123.45	2
"Hart, W."	"65 Lark Rd."	-43.00	3

{Person}			
name	address	balance	{Person}/
"Smith, A."	"20 Apple Rd."	-2.50	0

Order				
num	purchaser	item	quantity	Order/
1	(0,0)	0	5	0
2	(0,0)	6	10	1
3	(0,2)	5	3	2
4	(0,3)	3	5	3
5	(0,2)	2	2	4
6	(0,2)	7	8	5
7	(0,3)	3	3	6
8	(0,0)	0	2	7
9	(1,0)	5	2	8

Figure 3.7 Structure Coding Example (III)

In this third example no data is repeated and both members and non-members can be differentiated.

Summarizing the concepts illustrated by the previous three examples, structure coding allows the database designer to control how structures are coded and how codes are generated. Essentially, the designer specifies the intersections in the semilattice through type declarations. Three different approaches were demonstrated. Which approach is selected in practice depends on:

- the meaning of the data, and
- the desired efficiency.

Note that for purposes of illustration the HVFC database is quite simple. Perhaps the easiest way to handle the non-member problem is to suitably rename the Member

structure and add a new boolean attribute indicating whether each individual is a member. A boolean attribute achieves the same effect as the disjoint union tag in figure 3.7 but does not result in the creation of an internal relation.

3.5. Other Coding Schemes

Previous semilattice work [Arm84, Sin85, Arm86] has suggested using codes where part of the data referred to is duplicated or contained in the code itself. This is similar to relative coding where the uncoded attributes are concatenated to form part of the code. The ultimate goal of these coding schemes is to reduce disk accesses since certain operations (e.g. selection) could be performed on the codes without accessing the actual data.

The problems with coding schemes that embed data in the code are the same as with relative coding. Namely, the two requirements of coding schemes (R1) and (R2) are satisfied but all three properties can be violated, especially as a result of modifications. Except for unchangeable attributes, codes containing data should not be used unless the overhead of updating references following modifications can be tolerated. A difficult challenge is to design a general range preserving code similar to relative coding but which does not contain embedded data values and thus satisfies all the coding scheme properties.

A final potential coding scheme is called differential coding. Here, the value v of an entity is expressed as (v_c , v_d) where v_c is a common part and v_d is the difference of v from v_c. The common and difference parts are coded for all values. The code for a given value v is then the pair of codes (v_c' , v_d'). This coding scheme may eventually allow efficient selection operations but has not yet been developed to the point where the scheme can be thoroughly evaluated.

3.6. Comparison of Coding Schemes and Discussion

A summary of how well each of the three semilattice coding schemes meets the requirements and properties of coding schemes in contained in table 3.1 below.

Requirement or Property	Absolute Coding	Relative Coding	Structure Coding
R1	Y	Y	Y
R2	Y	Y	Y
P1	Y	N	Y
P2[1]	Y	Y	Y
P3	N	N	Y

[1] Duplicates not permitted.

Table 3.1 Coding Scheme Comparison

Each of the three schemes satisfies both requirements (R1) and (R2) but each satisfies the properties with varying degrees of success. Relative coding or any other scheme that embeds data in the code will not satisfy (P1) as already discussed.

Uniqueness (P2) is satisfied by all schemes but only when the database does not permit duplicate tuples or structures. To guarantee uniqueness when duplicates are allowed involves scanning for duplicate codes and then recoding following modifications. Since the recoding overhead will usually be prohibitive it may be better to allow (P2) to be violated. Operations can then only utilize codes if a list of aliases is maintained to identify different codes for the same value. Another option is to incrementally recode the database during idle periods. It should also be noted that (P2) is satisfied by codes that change with time provided uniqueness is maintained.

Structure coding is the only scheme which satisfies property (P3) since it eliminates repeated values. For applications such as CAD/CAM where the potential for redundancy is great due to multiple versions and extensive use of type constructors, satisfying this property is a significant advantage. As was illustrated, structure coding

can be directed by judicious use of the data definition facilities of the Conceptual Language.

There are three other issues that ultimately affect all coding schemes. We discuss these below in the context of structure coding.

3.6.1. Reuse of Codes

Recall that requirement (R2) states a code should be as compact as possible. Thus, when integer codes are used, we would like to reuse codes from deleted structures. That way, codes will not continue to grow as structures are deleted and new ones inserted. Reusing codes is easily accomplished in practice by keeping a free list of unused codes from deletions. When a new structure is inserted the first code on the free list is assigned if the list is not empty, otherwise a new code is generated. This method guarantees the largest code in the database can be no greater than N where N is the maximum number of objects ever held in the structure in the past.

However, as pointed out by Date [Dat83], codes should not be reused in case old archival data must be restored. This is why permanently assigned codes, or surrogates, are used in some models. To make it possible to restore archival data and reuse codes, an acceptable solution is to prevent a code from being reused immediately following the deletion of the corresponding structure. Instead, the code is held for a given expiry period before being placed on the free list. Introducing an expiry time is relatively simple and the expiry period can be dependent on the application. However, the notion of an expiry time requires additional bookkeeping to manage codes.

3.6.2. Modifications to Referenced Structures

There is some question as to how strict the model should be in permitting modifications to referenced (or nested) structures. When a referenced structure is modified indirectly from another entity the meaning of the referenced data changes for all structures referencing it. The problem is similar to the update problem for non-first-normal-form relations discussed by Arisawa [AMM83] where the actual updates "happen all over the database". To illustrate, the Member structure in figure 3.4 can be modified indirectly through the Order structure. This occurs because the purchaser attribute is of type Member and thus coded internally in the Order representation.

modify orders with orders.purchaser.balance = orders.purchaser.balance * 1.05

This feature is again related to the semantics of the data definition and is a powerful property of the Conceptual Language. Since such features are required for many applications it may be beneficial to include a database directive to limit modifications to base structures only.

3.6.3. Deletion of Referenced Structures

A related problem is deletion of referenced structures. What happens when a structure is deleted when that structure references another structure used elsewhere? Many database implementations have recognized this problem. The IPIP DBMS [JSW83] provides three deletion directives:

- delete all members when owner deleted,
- delete owner when last member deleted, and
- reject deletion when ownerless members would result.

The directives are basically predefined integrity constraints for avoiding the deletion problem. While these directives are somewhat satisfactory, a preferred solution for the semilattice model is to remove the problem from the conceptual level entirely.

When the database designer specifies shared structures with the Conceptual Language, the base structures are always maintained provided at least one reference exists in the database. This means ownership information, derived from the data definition, must be kept internally to record how structures reference one another.

To summarize, this chapter has proposed a new coding scheme called structure coding. The advantages of structure coding are as follows.

1. The resulting codes are very compact.
2. The use of the semilattice intersections is maximized.
3. Less storage is required than for absolute and relative coding.
4. Operations with codes are possible such as checking for duplicates, union, difference, and certain selections.
5. The coding scheme is uniform for tuples, sets, and sequences.
6. No recoding of codes is required because increasing the amount of indirection (coding depth) does not increase the code length.
7. The scheme does not depend on functional dependencies, multivalued dependencies, or first normal form.

Disadvantages of structure coding are:

1. Additional indirection may be introduced to eliminate repeated data.
2. The codes are not range preserving, limiting the selection operation with codes to selection for equality and inequality.

Chapter 4

Experimental Results and Evaluation

Chapter four presents some experimental results for the structure coding scheme and the semilattice model in general. First, the semilattice prototype is discussed including some limitations present in the current implementation. Efficiency of the model is evaluated next. Results for storage requirements are presented and an informal discussion of performance is given. Finally, the chapter concludes with an investigation of codes in distributed database applications. In particular, transmission costs and query optimization are examined.

4.1. Semilattice DBMS Prototype

The complete semilattice DBMS prototype implementation consists of two parts: the $C-B^+-tree$ software described by Singh [Sin85] and the conceptual level software presented here. A brief overview of the implementation is provided below since the prototype is used for some of the results in following sections.

4.1.1. Implementation

The semilattice prototype implementation was developed at the Computing Science department at the University of Alberta and runs under UNIX† 4.2BSD. The database operates as an interactive system which accepts user declarations and queries and then performs the requested action. In total, the conceptual level software comprises approximately 3600 lines of C source code which is divided into four modules. Appendix A2 contains the source code listing for each module.

1. Parser (parser.y).

† UNIX is a trademark of Bell Laboratories.

All syntax checking of user input is handled by the parser which consists of a yacc(1) specification of the Conceptual Language plus error recovery and formatting routines. Input to the parser is Conceptual Language statements and output is a reduced intermediate language. Syntax errors detected during the parse result in an error message identifying the problem. Further processing of the statement is aborted when a syntax error occurs.

2. Interpreter (interp.c).

The interpreter module performs semantic checking on the output from the parser. For data definition statements, the interpreter checks that the type or variable is not already defined, and if not, builds a new symbol table entry. No further processing is required for data definitions so control is returned to the parser. Data manipulation statements, or operations, are first type checked and then the semantics for the particular operation are verified. Examples of semantic errors that may occur include the use of undefined variables or illegal operations (such as multiplying two strings). Type or semantic errors result in an error message and control again returns to the parser.

3. Operators (oper.c).

Data manipulation statements which are syntactically and semantically correct are executed by the operators module. This module contains functions for retrieving, structuring, and coding data items plus routines for each major semilattice operation (insert, delete, modify, union, difference, product, project, select, and print). Errors are not normally expected at this stage but when errors do occur it is due to either the failure of a lower level routine or possibly a corrupt database.

4. Stubs (stub.c).

The final optional module contains Find, Insert, and Delete stubs which simulate the lower level $C-B^*-tree$ software. If this module is included the conceptual

level software can be run standalone with structures stored in flat files by UNIX system calls. This module is omitted when the $C-B^+-tree$ software is used.

4.1.2. Limitations

Since the current semilattice DBMS implementation is a prototype it possesses several limitations due to features which are not supported or are only partially supported. Several enhancements are also required for more flexibility. Furthermore, the current Conceptual Language definition provides only the basic data definition and manipulation constructs and is thus incomplete as a full specification or database programming language. However, the current language definition is more than adequate for prototype purposes: All of the limitations listed below are presently of little significance as the main goal of the prototype is to prove the fundamental concepts of the semilattice model: the high level conceptual model and the use of semilattice codes.

Three constructs are missing or incomplete in the prototype. Constraints are not supported although the Conceptual Language provides a means for their definition. Handling constraints involves maintaining the constraint information in an intermediate language form associated with each type or structure and then checking for constraint violations during operations. The null tag is also currently not supported but can be easily added to existing tag information maintained with attribute values. The final incomplete construct is that set operations (union, difference, and product) are currently restricted to tuple schemes even though any generic type should be allowed. Completing the set operations involves defining the operations for the remaining types.

During the course of the development several enhancements to the semantics of the Conceptual Language which would be very beneficial were noted. Each of these generally increases the flexibility of the model and makes the language easier to use.

1. Type coercion.

Coercion between compatible types is a useful extension. Integer to real coercion with Pascal semantics and real to integer coercion with rules for rounding are straightforward. Sequence to set coercion is also useful provided duplicate elements are removed. Sets can be coerced to sequences but because this coercion adds information the order of the elements in the resulting sequence is undefined.

2. Automatic typing.

When the result of an operation is assigned to a new variable the Conceptual Language requires that the target variable be defined. For example, to select overdrawn members from the HVFC Member structure:

```
Overdrawn_name = ( name : string , address : string , balance : real )
overdrawn_names : { Overdrawn_name }

overdrawn_names := members where members.balance < 0.0
```

Since the type of the result of an operation is always well defined in the Conceptual Language, the declarations in the above example could be eliminated. When such *automatic typing* is used, undefined target variables are assigned the type of the result. With this extension, the above example simplifies to a single statement.

```
overdrawn_names := members where members.balance < 0.0
```

Automatic typing eliminates extraneous declarations and makes it easier to manipulate intermediate results.

3. Context sensitive parsing of qualified names.

Whenever qualified names (i.e. parts of structures such as members.balance) are used, the full variable name, attribute names, and tag (if any) are required by the Conceptual Language. In many cases such a long specification is unnecessary since at least part of the name can usually be derived from the context. By making the

Conceptual Language context sensitive when qualified names are used, the overdrawn members example above reduces to

overdrawn_names := members where balance < 0.0

where balance is expanded to members.balance. With this extension, fully qualified names would still be required in ambiguous situations, otherwise an error would result.

4.1.2.1. Data Packing

In the current prototype, data is stored inefficiently as ASCII strings using the UNIX "stdio" functions. This is done primarily to assist debugging. Packing and unpacking of data is contained in two functions, Pack and Unpack, so that the packing procedure can be easily changed at any time.

A more efficient packing format which supports all semilattice simple and complex types is shown in figure 4.1 in its general form. Figure 4.2 illustrates how actual values are stored. The format is very flexible allowing for future optimization such as special case handling, for example, sequences of booleans, or the addition of compression algorithms discussed in the next section.

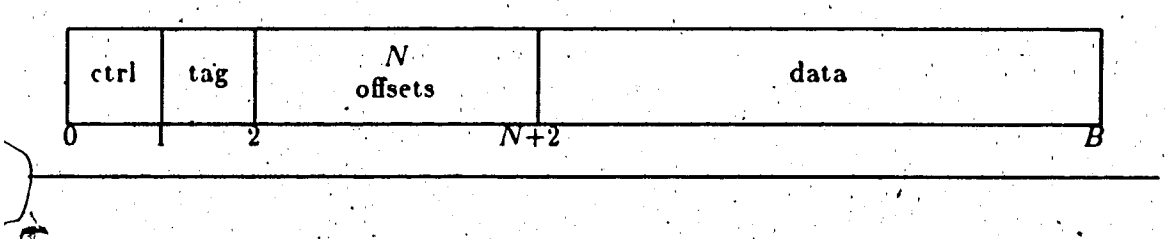

Figure 4.1 Packing Block Format

The format packs data into blocks of B bytes where $3 \leq B \leq 256$. The control (ctrl) byte contains the number of offsets minus one (bits b_0 - b_2), a "more" bit indicating whether another block follows (bit b_3), and the offset number containing the next block

offset (bits b_4 - b_6) if the more bit is set. A maximum of eight data items can be stored in the block using the format. If another block follows only seven data items are stored since the eighth offset is to the following block, not to a data item.

The tag byte indicates whether data item i is tagged in bit b_i. Bit b_i is zero for untagged data or the null tag and one for properly tagged data. Following the tag byte are N offsets into the block where $1 \leq N \leq 8$. Each offset i indicates where the i^{th} data value is located within the block.

Simple data values are stored using the following sizes.

- Integer: 2 bytes (can also support 4 byte long integers).
- Real: 4 bytes (can also support 8 byte double precision).
- Boolean: 1 byte.
- String: 1 byte per character terminated by a null.

Tuples are stored simply by packing the attribute values into one or more blocks in the order the attributes appear in the tuple scheme definition. The number of attributes is always known from the tuple scheme definition. Sets and sequences of elements are packed into as many blocks as required with the next available offset after the packing set to null to indicate the end of the set or sequence. Tagged values where the corresponding tag bit is set are stored as a tag terminated with a null character followed by the value.

Figure 4.2 shows how a tuple from the HVFC Member structure and a set from the {Priced_item} structure are packed.

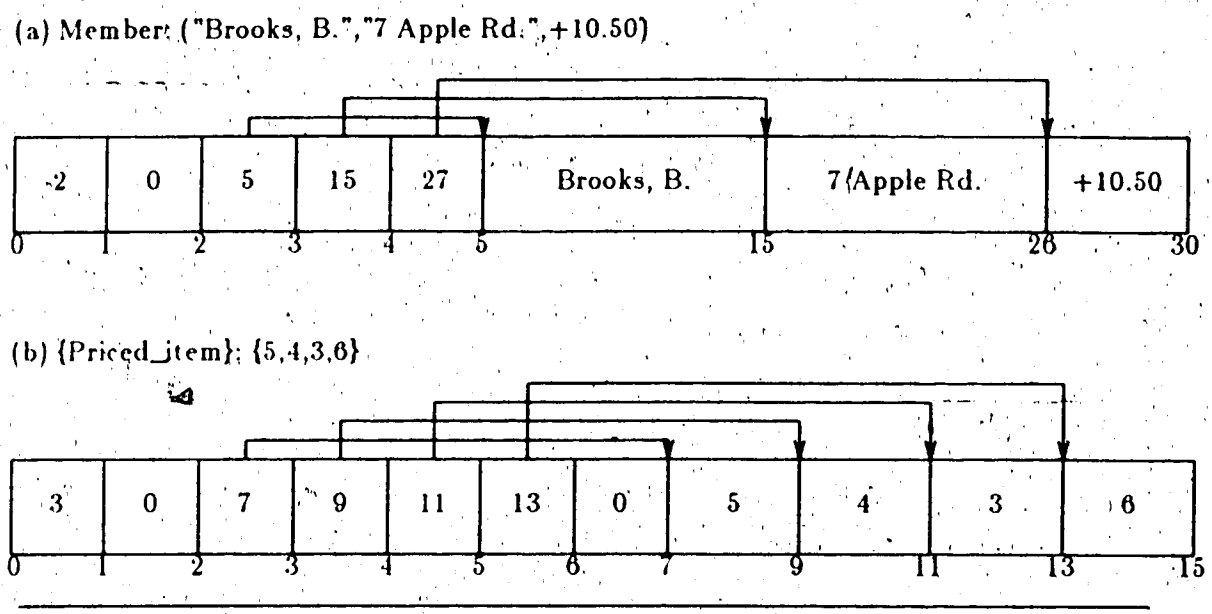

Figure 4.2 Packing Example

Note the use of the null offset in byte six of the {Priced_item} block to terminate the set.

This packing procedure is completely general and can also support long integers, double precision reals, and compression. The only inherent restriction is that a string value cannot span blocks and is thus limited to 256 − 4 = 252 bytes. If it is necessary to support longer string values, the restriction can be overcome by using two byte offsets.

4.1.3. Data Compression

Although coding is used in the prototype, no further attempt is made to compress data values of simple types. Several well known and easily implemented compression techniques could be employed. For integers, bit vectors and differences can be used. For both integers and reals, leading and trailing zeros can be eliminated. Floating point numbers can be truncated or represented as two binary numbers where

appropriate [Wie77]. Character strings can be compressed in many ways including through such techniques as abbreviation, variable length strings, deleting blanks, and replacement [Wie77]. Slightly more sophisticated methods for compressing alphanumeric data are available including minimum bit compression, character repeat suppression, common phrase suppression, n-gram coding, and dictionary recoding [Lyn85]. Most commercial databases provide at least some primitive methods for data and index compression.

Front and rear compression [Dat81] can be used for sequential processing applications or for non-integer codes (such as with relative coding). Huffman coding [Sam70] can be used when character frequencies are known or can be accurately estimated as is the case for the English alphabet. One commercial DBMS software package, called SHRINK, provides Huffman coding as a compression technique for IMS databases [Lyn85]. A compression ratio of up to 2:1 is claimed.

4.2. Storage Efficiency

Since the semilattice model uses short codes instead of keys, one expects an efficient use of storage. To quantify the storage efficiency of the model, several experiments were performed comparing INGRES storage requirements with those of the semilattice prototype. INGRES version 7.10 [Eps77, Woo82] (UCB version) and the semilattice prototype with structure coding were used. For each test, 100 tuples were randomly generated using the pseudo random number generator of UNIX (random(3)). The generator was seeded with the same initial value for each test. Tuples schemes from the HVFC example database were used.

Figure 4.3 compares INGRES and prototype storage requirements for the Member structure. Recall that since the Member tuple scheme attributes are all simple types, no coding is required and the structure represents a traditional relation. This first test

is necessary to determine whether major differences exist between the systems and provides a basis for further comparison.

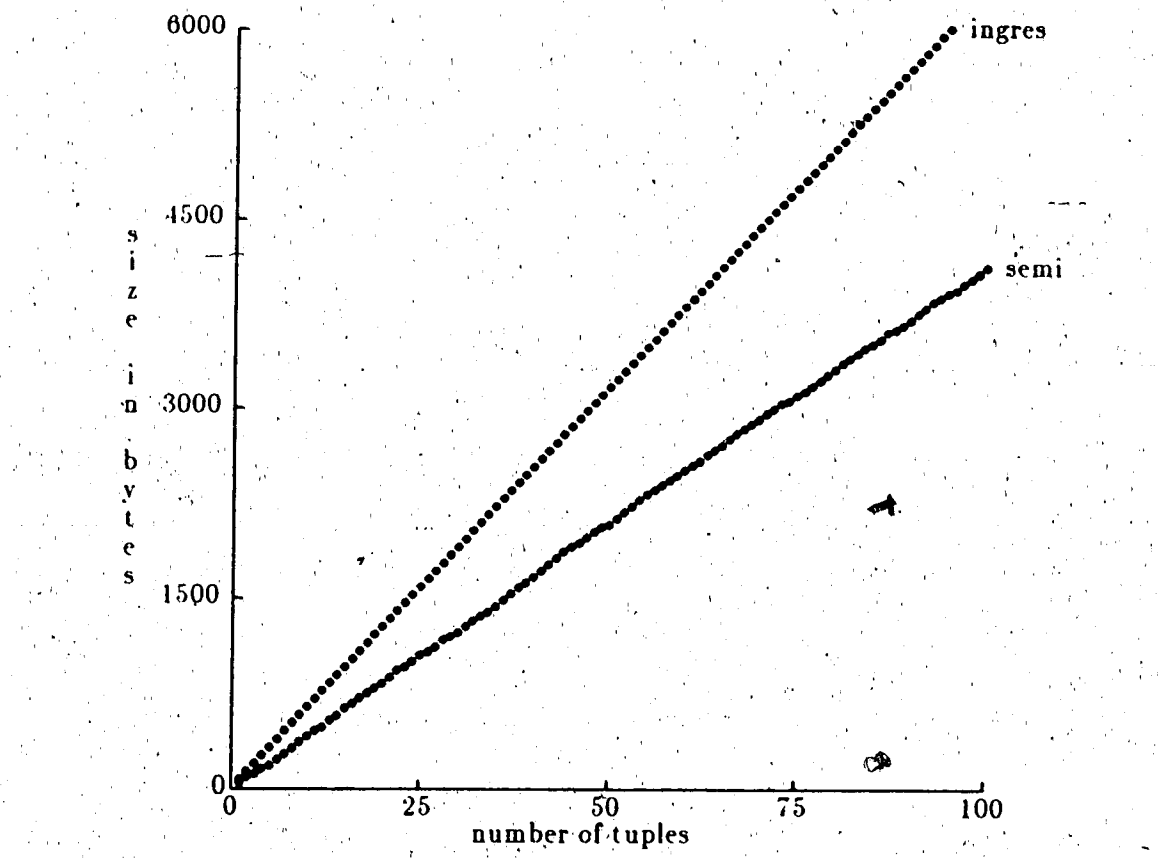

Figure 4.3 Storage Requirements Comparison (I)

As shown above, INGRES consumes more storage even for a typical relation. After 100 tuples, the structure under INGRES required 6300 bytes while under the semilattice prototype only 4096 bytes was used which represents a saving of approximately 35%. This difference can be attributed to the fact that the prototype stores variable length strings while INGRES stores fixed length strings with padding where necessary. In fact, if at least two attributes in an INGRES relation are declared to be c255 (string of length 255), each tuple will fill a page (1024 bytes) even if the values of the long strings are just single characters. There is no provision to declare variable length strings in

INGRES although it is possible to load and unload the database using a variable length string format. Newer releases of INGRES, such as the commercial RTI version, include many improvements to overcome most problems of this type.

The next figure, 4.4, illustrates the effect coding has on storage requirements. Here, storage for the Order structure is measured. In the INGRES relational database the tuple scheme is

```
Orders = ( num : integer , purchaser : string , item : string ,
           quantity : integer)
```

where the purchaser and item attributes are keys in the members and goods relations respectively. With structure coding, the purchaser and item attributes are coded.

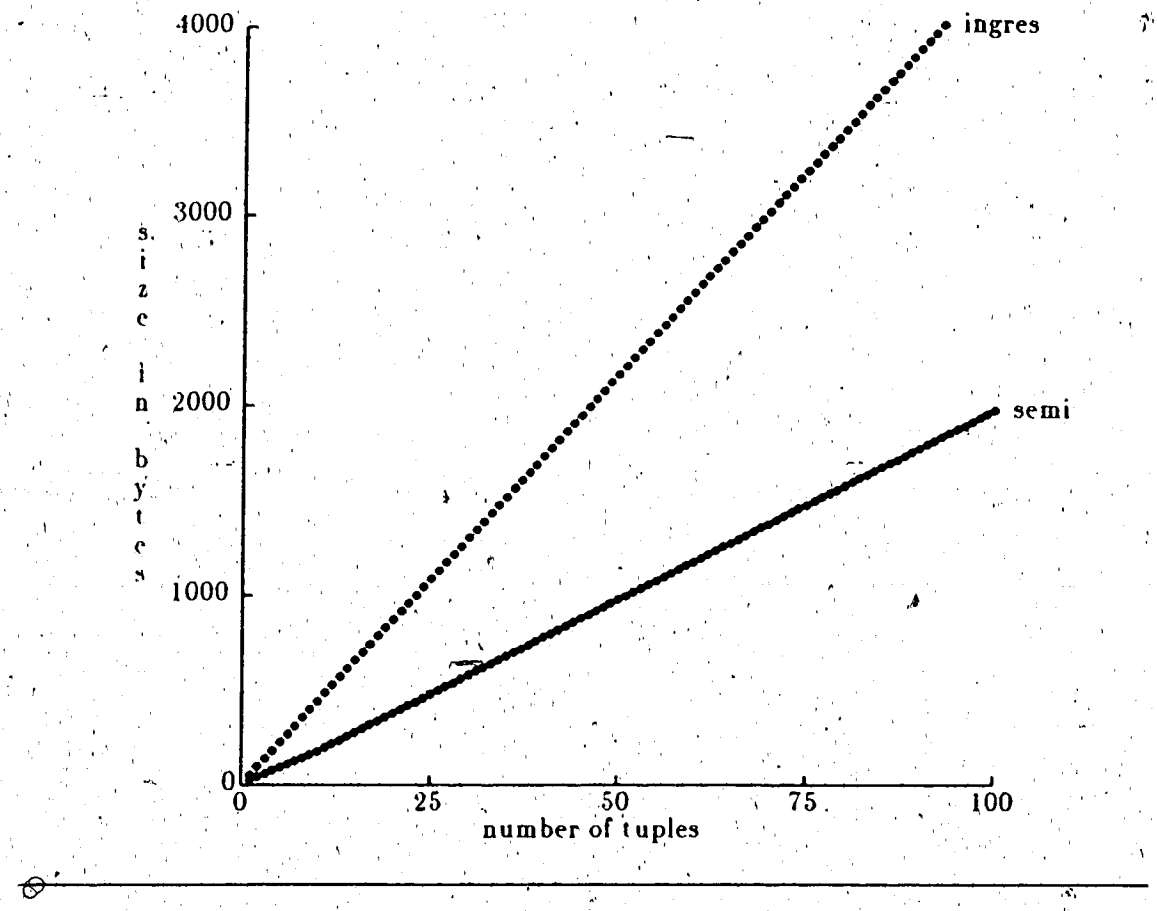

Figure 4.4 Storage Requirements Comparison (II)

The storage savings due to coding are dramatic: for 100 tuples, INGRES required 4300 bytes while the prototype required only 1961 bytes. This represents a saving of approximately 54%. Notice that the slope of the semilattice curve in the figure changes very slightly near the 10 tuple point on the horizontal axis. This is the point at which two digit codes become required in the prototype for the coded attributes resulting in longer tuple values on average.

Table 4.1 summarizes the storage requirements for the remaining structures in the HVFC database. The Priced_item structure contains only simple types and is represented in the same manner for both databases. The Supplier tuple scheme under INGRES is declared as

Supplier = (name : string , address : string , item : string , price : real)
suppliers : { Supplier }

where item and price form the key for the goods relation. Recall that in the semilattice case the Supplier structure has the stock attribute coded.

Structure	Storage (bytes)		Saving (%)
	INGRES	Semilattice	
Member	6300	4096	35.0
Priced_item	2400	2418	0.0
Orders	4300	1961	54.4
Supplier	7900	3164	59.9

Table 4.1 Storage Requirements Summary

Storage savings with the prototype vary from a low of 0.0% for the simple Priced_item structure to a high of 59.9% for the Supplier structure. The most noticeable saving occurs not in storing simple structures but for structures where one or more attributes are coded.

The storage efficiency of the prototype is directly due to the semilattice condition in which intersections in the lattice are stored only once. Structure coding is the vehicle by which the condition is achieved in the implementation.

4.3. Performance Issues

Besides the question of storage efficiency, the performance issue must also be considered in any evaluation of the semilattice model. This section discusses three related performance topics: disk accesses, performance structures, and the use of a knowledge base for optimization purposes.

4.3.1. Disk Access

A detailed evaluation of the disk access patterns of the semilattice model is beyond the scope of this thesis. A proper evaluation must consider the access method used whether it is B-trees, $C-B^{*}-trees$, one of the many indexed schemes, hashing, or some other method. The effects of the underlying operating system file system must also be considered unless the database completely bypasses the operating system and instead implements its own specialized file system. Insertion and deletion costs must be examined in addition to several different kinds of retrieval costs.

The conceptual level of the semilattice model is independent of the access method used. We can, however, provide an informal discussion of the disk access question from a conceptual point of view. The use of semilattice codes will tend to reduce disk accesses for some operations since the operation algorithms can utilize codes thus saving accesses to the actual data. Since codes are short, more codes, and thus more information, can be packed into a block and transferred in a single disk block access. Sacks-Davis has shown coding schemes perform well for the partial match retrieval problem [Sac85] which is very similar to a selection operation.

On the other hand, we know normalization in the relational model tends to produce less efficient designs because wide relations are usually broken up into many smaller relations [Ken83a]. The same is true for the semilattice model because by separating repeated data, additional indirection is introduced (similar to normalization) and this leads to increased disk accesses. However, the database designer can control indirection to some extent with structure coding through judicious data definition.

A detailed study is required to determine which access method is best suited for the semilattice model. To date, the net effect of all the various parameters on performance remains unclear. Instead of $C-B^{*}-trees$, a storage structure such as K-D-B

trees [Rob81] may be more efficient. Since semilattice codes form a one-to-one mapping with the data, a storage structure capable of efficiently exploiting the mapping is required to facilitate encoding and decoding. For example, doubly indexed $C-B^{+}-trees$ may be appropriate.

4.3.2. Performance Structures

Should performance turn out to be a problem for certain applications, it is still possible to maintain special structures to keep disk accesses to a minimum. *Performance structures* are the usual semilattice internal structures but do not contain any coded values, that is, all values are decoded. Although less information is retrieved with each disk block access, it is no longer necessary to make additional accesses to retrieve coded data. Performance structures may be broadly defined as any structures that go beyond the mere storage of information to improve performance. Examples of performance structures include non-first-normal-form relations such as those proposed in Schek [ScP82] and Arisawa [AMM83], joined relations, and uncoded structures.

When performance structures are incorporated into the semilattice model, the structures are maintained automatically by the database. The database administrator must identify the performance structures using a special directive. All semilattice operations work the same for performance structures except the encoding or decoding stage is omitted. Whether performance structures are being used in practice is transparent to the average user except for possibly faster query response times for these structures.

4.3.3. Knowledge Base

Previous semilattice work [Bob84,Sin85] has suggested an adaptive component for the model that monitors and optimizes the database using statistics gathered and retained in a knowledge base. Various meta-data can be stored in the knowledge base including:

- constraints and dependencies,
- usage and access patterns,
- current physical organization of the data, and
- reorganization decisions.

A feasibility study of the adaptive component [Sin85] concluded such a system was indeed possible. It was noted that other researchers are currently proceeding in this direction.

Management of performance structures can also be effectively handled by the adaptive system. In monitoring the database access patterns, the adaptive system can identify heavily referenced structures which are candidates for performance structures. If the decision is taken to do so, a candidate structure can be converted to a performance structure automatically by the database. Should usage of the structure subsequently decrease, the adaptive system can change the performance structure back to a standard semilattice structure. Safeguards should be included in the adaptive system to prevent oscillations in which structures are repeatedly converted between representations.

4.4. Distributed Database Applications

This section examines how transmission costs in a distributed database environment can be reduced when the semilattice model is used. In all cases, *cost* is a measure of the amount of data in bytes that must be transmitted in order to satisfy a query. Various network and queuing delays are not considered here since in most cases network bandwidth will be the limiting factor.

4.4.1. Use of Codes

Existing distributed relational database systems use tuple identifiers to assist in manipulating fragmented relations. For example, System R^* uses TIDs to solve problems related to the deletion of tuples in vertically fragmented relations [Ull82]. Semilattice codes are similar to tuple IDs and also uniquely identify relations. Since structure codes are small integer numbers, they can be used to substantially reduce transmission costs between sites. The cost to transmit a relation R between sites is given in Ceri and Pelagatti [CeP84] to be

$$C = C_0 + C_1 * size(R) * card(R) \qquad (4a)$$

where,

C_0 = fixed cost of initiating a transmission between sites,
C_1 = network unitary transmission cost,
$size(R)$ = average size of a tuple in R in bytes, and
$card(R)$ = number of tuples in R.

If codes are used instead,

$$C_c = C_0 + C_1 * size(R_c) * card(R) \qquad (4b)$$

where,

$size(R_c)$ = average size of a code in R in bytes.

Since $size(R_c) < size(R)$ it follows that $C_c << C$. For integer structure codes,

$$size(R_c) = \left\lceil \frac{\log_2 card(R)}{8} \right\rceil \tag{4c}$$

so that (4b) becomes:

$$C_c = C_0 + C_1 * \left\lceil \frac{\log_2 card(R)}{8} \right\rceil * card(R) \tag{4d}$$

To illustrate using an example, suppose the Member structure in the HVFC database has size(name) = 25, size(address) = 30, and size(balance) = 8. Then, taking $C_0 = 0$ and $C_1 = 1$ with 1000 tuples gives:

$$C = 0 + 1 * (25 + 30 + 8) * 1000 = 63000$$
$$C_c = 0 + 1 * \left\lceil \frac{\log_2 1000}{8} \right\rceil * 1000 = 2000$$

Figure 4.5 shows how C and C_c grow with $card(Member)$. The slope of the C_c curve changes at $card(Member) = 256$ because at this point two byte codes are required.

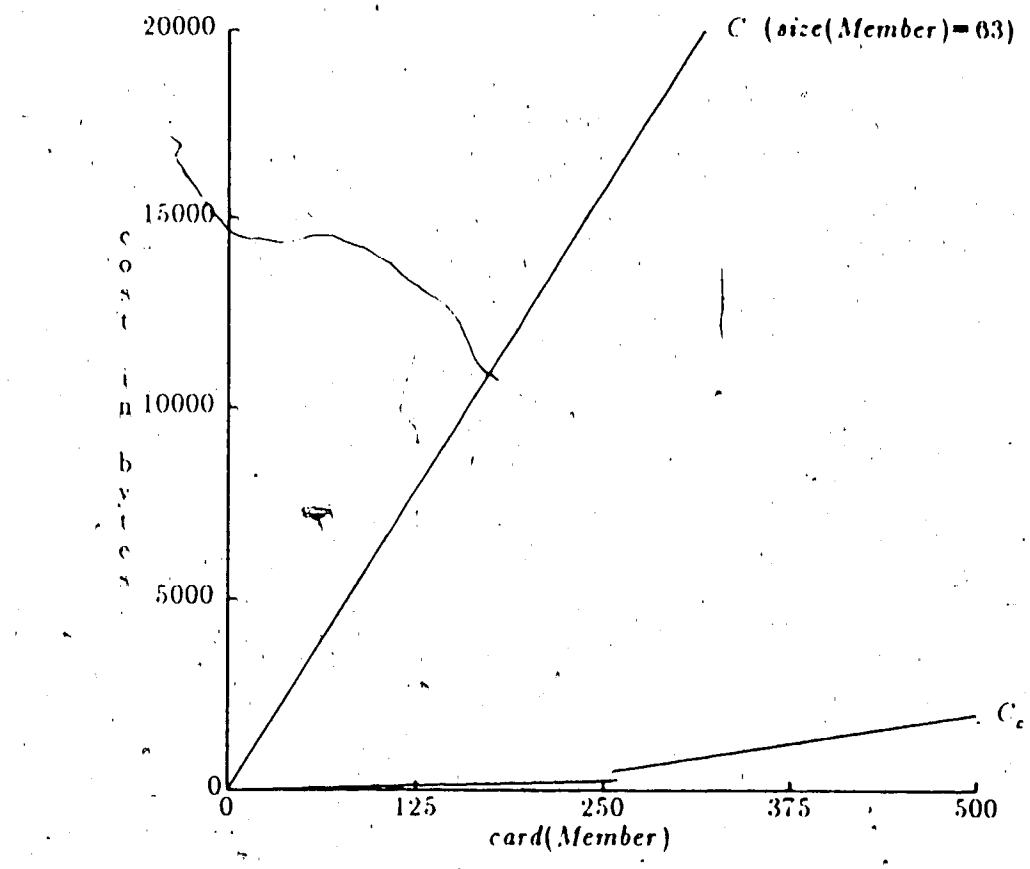

Figure 4.5 Transmission Cost Versus Cardinality of a Structure (1)

The transmission cost using codes is significantly less; the saving is even greater for wide relations. More importantly, for a fixed $card(R)$, note that C_c is a constant for all values of $size(R)$ while C increases with $size(R)$. C_c is always less than C except possibly for very narrow relations with many tuples where $size(R) < size(R_c)$.

One method to maintain semilattice codes is to use encoding and decoding files as outlined by Singh [Sin85]. In distributed applications these files can also be used directly to interpret codes transmitted from remote sites. When structures are fragmented, no additional disk overhead is incurred by transmitting codes since the encoding and decoding files must be replicated at each site in any case. If separate encoding

and decoding files are used for each fragment then only the decoding files must be replicated at the remote site when codes are transmitted.

To illustrate this point, suppose the Member structure is horizontally fragmented and allocated to two sites "A" and "B". Case (1) below shows the one encoding and one decoding file required at each site when uniform coding is used between fragments. In case (2), each fragment is coded independently. This case shows the additional decoding file required for codes transmitted from the remote site.

1. site "A": site "B":
   Member encoding file — Member encoding file
   Member decoding file — Member decoding file

2. site "A": site "B":
   $Member_A$ encoding file — $Member_B$ encoding file
   $Member_A$ decoding file — $Member_B$ decoding file
   $Member_B$ decoding file — $Member_A$ decoding file

Case (1) may incur a high overhead in dynamic databases since the replicated encoding and decoding files must be updated at each site. The second case requires fewer updates since only the decoding file is replicated. Storage costs are approximately equal in both cases since only half the space is required for the local encoding and decoding files in case (2) but an additional file is present (assuming both fragments are of equal size).

4.4.2. Query Optimization with Codes

Query optimization algorithms that make use of the semijoin, such as that used by SDD-1 [Ber81], can also benefit from the transmission of codes. The cost of a semijoin over relations R and S on attribute B is given by Ceri [CeP84] to be

$$C_{sj} = 2C_0 + C_1 * (size(B) * val(B(S)) + size(R) * card(R')) \qquad (4e)$$

where,

$size(B)$ = average size of join attribute B in bytes,
$val(B(S))$ = number of distinct values of attribute B in S, and
R' = semijoin result $R \ltimes S$.

When result R' is coded,

$$C^1_{sj} = 2C_0 + C_1 * (size(B) * val(B(S)) + size(R_c) * card(R')) \quad (4f)$$

Substituting (4c) into (4f) gives:

$$C^1_{sj} = 2C_0 + C_1 * (size(B) * val(B(S)) + \left\lceil \frac{\log_2 card(R)}{8} \right\rceil * card(R')) \quad (4g)$$

If the join attribute is not dense and thus also coded,

$$C^2_{sj} = 2C_0 + C_1 * (size(B_c) * val(B(S)) + size(R_c) * card(R')) \quad (4h)$$

where,

$size(B_c)$ = average size of a code for B in bytes.

Substituting (4c) into (4h) yields:

$$C^2_{sj} = 2C_0 + C_1 * \left(\left\lceil \frac{\log_2 size(B)}{8} \right\rceil * val(B(S)) + \left\lceil \frac{\log_2 card(R)}{8} \right\rceil * card(R') \right) \quad (4i)$$

The benefits from C^1_{sj} and C^2_{sj} can be substantial. To illustrate, suppose the Order structure is vertically fragmented and allocated to two sites as follows.

$Order_A$ = (num : integer , item : Priced_item , quantity : integer)
$Order_B$ = (num : integer , purchaser : Member)

$orders_A : \{Order_A\}$
$orders_B : \{Order_B\}$

For 1000 orders and again taking $C_0 = 0$ and $C_1 = 1$ we have:

$$C_{sj} = 0 + 1 * (4 * 1000 + (4+1+1) * 1000) = 10000$$

$$C^1_{sj} = 0 + 1 * (4 * 1000 + \left\lceil \frac{\log_2 1000}{8} \right\rceil * 1000) = 6000$$

If the num attribute is also coded:

$$C^2_{sj} = 0 + 1 * \left(\left\lceil \frac{\log_2 1000}{8} \right\rceil * 1000 + \left\lceil \frac{\log_2 1000}{8} \right\rceil * 1000 \right) = 4000$$

However, suppose the semijoin algorithm is not used and instead the entire coded structure is transmitted. Then using equation (4d),

$$C_c = 0 + 1 * \left\lceil \frac{\log_2 1000}{8} \right\rceil * 1000 = 2000$$

which is even less costly than the best semijoin. The reason for this result can be seen by comparing equation (4d) with any of the semijoin equations (4e), (4g), or (4i). Figure 4.6 compares the various costs for this example. Again, the discontinuities at the 256 tuple point are due to the change from one byte to two byte codes.

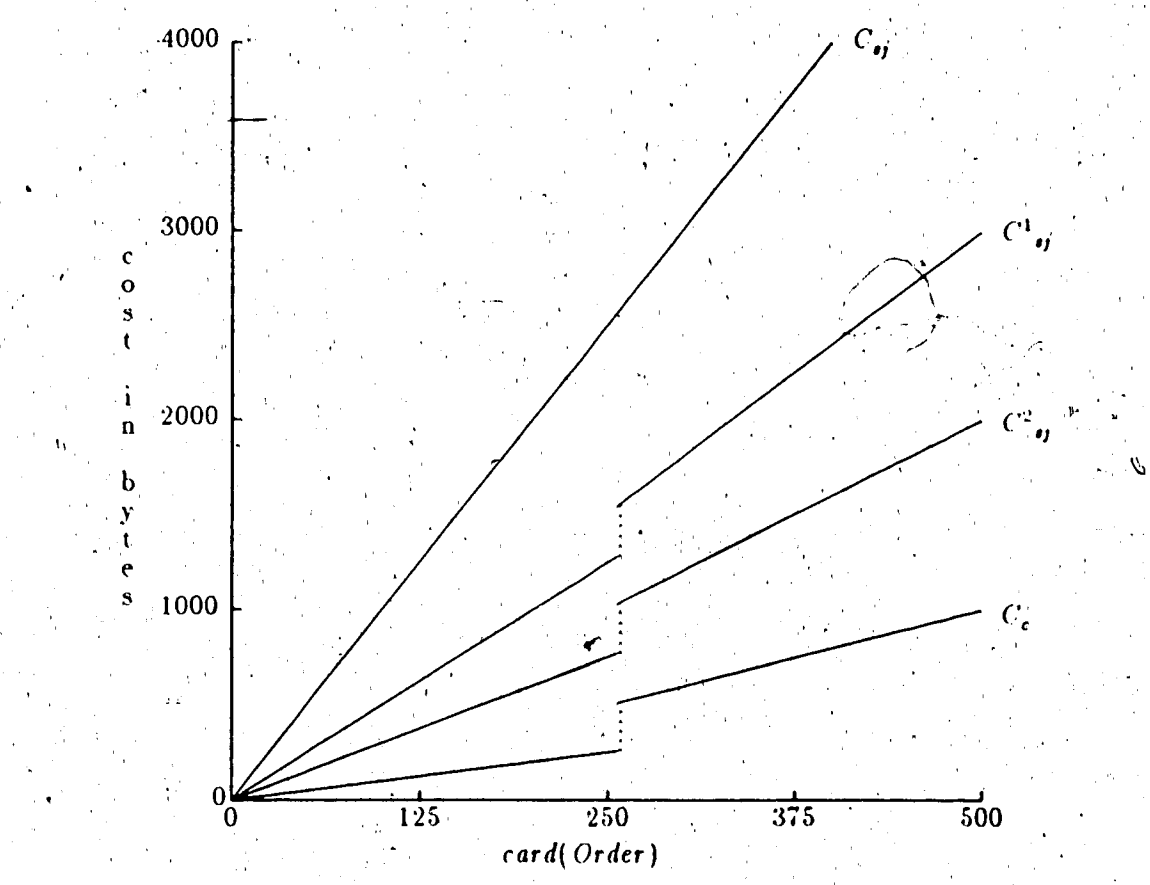

Figure 4.6 Transmission Cost Versus Cardinality of a Structure (II)

With the semijoin approach, two sets of transmissions are required: transmit all the join attributes, and transmit the reduced relation for use in the final join. Since the transmission cost of codes is quite low, it will often be the case that $C_c < C^1_{ij}$ as in the example here. The only time when this result does not hold is when very few tuples participate in the join.

Therefore, if codes are used, query optimization can be simplified by eliminating the need for semijoins when a majority of tuples are expected to participate in the operations. The disadvantage is the additional local decoding file as pointed out earlier.

Chapter 5

Conclusions

5.1. Summary

This thesis has investigated coding schemes, one of the fundamental concepts of the semilattice model of data. The major contribution of the study is a new proposed coding scheme called structure coding which performs better than previously known schemes. An analysis of the new coding scheme is supported by a working prototype implementation which incorporates structure coding.

5.2. Coding Schemes

In designing any coding scheme for the semilattice model it is very important that the requirements and properties of coding schemes stated here be satisfied. Firstly, the scheme must provide a method to refer to common data (R1). Second, the scheme must provide a compact method of reference (R2). Third, the scheme should provide a code which is unaffected by changes to the underlying data object (P1). Fourth, the scheme should provide a single unique code for each entity (P2). Fifth, and lastly, the scheme should provide a method to eliminate repeated values in representing objects (P3).

Three coding schemes were compared using these five criteria. All schemes satisfied the two requirements but each satisfied the properties with varying degrees of success. Relative coding performs least satisfactory since properties (P1) and (P3) are not met. The major disadvantage of relative coding is that modification of uncoded attributes changes the code. This limits the applicability of relative coding to values which are static or not changeable. Absolute coding performs quite well and only fails to satisfy property (P3). For simple database applications this is not a significant problem.

Structure coding is a new coding scheme in which structures (sets, sequences, and tuples) are coded rather than simply relations. The scheme may create additional internal relations when structures of the same type are embedded within other structures. Structure coding enforces the semilattice condition by eliminating all repeated data without loss of information. A method is provided as part of the data definition facilities whereby the database designer can direct structure coding by assigning different types to structures, and thus determine the semilattice intersections. Structure coding is the only one of the three coding schemes to satisfy all five evaluation criteria.

The semilattice prototype implementation was used to evaluate the effectiveness of structure coding. Storage requirements for INGRES and the prototype were compared using the HVFC example database. Storage savings with the prototype ranged from 0% to approximately 60% depending on the amount of coding in the structure. This limited evaluation supports the claim that the semilattice model is very storage efficient. Storage efficiency is a direct result of the semilattice condition which is realized by structure coding.

5.3. Directions for Further Work

This thesis, together with the work on the physical storage structures [Sin85], has laid the groundwork for further development of the semilattice model. The two studies have shown that the model is indeed viable, both at the conceptual and physical levels, as witnessed by the current prototype implementation. However, much additional work is required to develop the semilattice model into a recognized data model.

A few secondary issues still remain to be addressed:

1. The adaptive component including the knowledge base and meta-data.

This component of the system requires investigation to determine the overhead

and performance gains associated with an adaptive database. A sufficiently general adaptive tool has applications for other data models besides the semilattice.

2. The physical access method to support the semilattice model.

   $C-B^+-trees$ may not be the optimal supporting data structure. This question should be reevaluated now that a working prototype is available.

3. The question of query optimization.

   Currently, the prototype does not include optimization and little work has been done in this area. It may be possible to achieve significant gains through optimization especially when performance structures are considered.

Aside from these smaller issues, there are two more significant questions requiring investigation. First, although this thesis has demonstrated the storage efficiency of the model, the performance efficiency in terms of disk accesses has yet to be shown. One of the goals of semilattice research has always been to reduce disk accesses by operating on codes containing useful information [Arm84, Arm86]. This is still possible to some extent with integer structure codes but maintaining informational codes for values that may change conflicts with property (P1). Codes have many positive and negative effects on performance but the net effect remains unclear at this stage. Measuring disk accesses is also further complicated by the physical access method and file system of the machine.

The other major area for further work is the high level conceptual model, the second fundamental concept of the semilattice model. The present Conceptual Language is rudimentary and incomplete and is thus suitable only for prototype purposes. Several extensions have already been suggested: coercion between types, automatic typing, and context sensitive parsing. A better method for specifying constraints, and possibly dependencies, is probably required as are programming language constructs. Ultimately, a complete database programming language along the lines of

Galileo [ACO85] will be needed to fully support the high level semilattice conceptual model.

References

[ACO85] A. Albano, L. Cardelli and R. Orsini, Galileo: A Strongly-Typed, Interactive Conceptual Language, *ACM Transactions on Database Systems vol. 10, no. 2*, (June 1985), pp. 230-260.

[AMM83] H. Arisawa, K. Moriya and T. Miura, Operations and the Properties on Non-First-Normal-Form Relational Databases, *Proceedings 1983 International Conference on Very Large Data Bases*, October 1983, pp. 197-204.

[Arm74] W. W. Armstrong, Dependency Structures of Data Base Relationships, *Proceedings 1974 IFIP Conference*, Amsterdam, 1974, pp. 580-583.

[ArM83] W. W. Armstrong and T. H. Merrett, The Semilattice Model: An Internal Data Model, Unpublished, Department of Computing Science, University of Alberta, Edmonton, Alberta, 1983.

[Arm84] W. W. Armstrong, A Semilattice Database System, Unpublished, Department of Computing Science, University of Alberta, Edmonton, Alberta, November 1984.

[Arm86] W. W. Armstrong, Outline of the Semilattice Data Model, Unpublished Draft, Department of Computing Science, University of Alberta, Edmonton, Alberta, February 1986.

[Ast76] M. M. Astrahan et. al., System R: A Relational Approach to Data Management, *ACM Transaction on Database Systems vol. 1, no. 2*, (June 1976), pp. 97-137.

[BaK85] D. S. Batory and W. Kim, Modeling Concepts for VLSI CAD Objects, *ACM Transactions on Database Systems vol. 10, no. 3*, (September 1985), pp. 322-346.

[BFH77] C. R. Beeri, R. Fagin and J. H. Howard, A Complete Axiomization for Functional and Multivalued Dependencies, *ACM SIGMOD International Conference on Management of Data*, 1977, pp. 47-61.

[Bee80] C. Beeri, On the Membership Problem for Functional and Multivalued Dependencies, *ACM Transactions on Database Systems vol. 5, no. 3*, (September 1980), pp. 241-259.

[Ber76] P. A. Bernstein, Synthesizing Third Normal Form Relations from Functional Dependencies, *ACM Transactions on Database Systems vol. 1, no. 4*, (December 1976), pp. 277-298.

[Ber81] P. A. Bernstein et. al., Query Processing in a System for Distributed Databases (SDD-1), *ACM Transactions on Database Systems vol. 6, no. 4*, (December 1981), pp. 602-625.

[Bla81] M. W. Blasgen et. al., System R: An Architecture Overview, *IBM Systems Journal vol. 20, no. 1*, (1981), pp. 41-62.

[Bob84] K. Bobey, The Semilattice Data Model: Conceptual Language General Specification, Unpublished, Department of Computing Science, University of Alberta, Edmonton, Alberta, 1984.

[BuF79] P. Buneman and R. E. Frankel, FQL - A Functional Query Language, *ACM SIGMOD International Conference on Management of Data*, 1979, pp. 52-58.

[COD71] CODASYL, *Data Base Task Group Report*, ACM, New York, New York, April 1971.

[COD78] CODASYL, *COBOL Journal of Development*, 1978.

[CFP82] M. A. Casanova, R. Fagin and C. H. Papadimitriou, Inclusion Dependencies and their Interaction with Functional Dependencies, *Proceedings ACM Symposium on Principles of Database Systems*, 1982, pp. 171-176.

[CeP84] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill Book Company, New York, New York, 1984.

[Cha81] D. D. Chamberlin et. al., A History and Evaluation of System R; *Communications of the ACM vol. 24, no. 10*, (October 1981), pp. 632-646.

[Che76] P. P. Chen, The Entity-Relationship Model: Toward a Unified View of Data, *ACM Transactions on Database Systems vol. 1, no. 1*, (March 1976), pp. 9-36.

[Cin78] Cincom, *OS TOTAL Reference Manual*, Cincom Systems, Cincinnati, Ohio, 1978.

[CCW85] B. G. Claybrook, A. M. Claybrook and J. Williams, Defining Database Views as Data Abstractions, *IEEE Transactions on Software Engineering vol. SE-11, no. 1*, (January 1985), pp. 3-14.

[CIW83] J. Clifford and D. S. Warren, Formal Semantics for Time in Databases, *ACM Transactions on Database Systems vol. 8, no. 2*, (June 1983), pp. 214-254.

[Cod70] E. F. Codd, A Relational Model for Large Shared Data Banks, *Communications of the ACM vol. 13, no. 6*, (June 1970), pp. 377-387.

[Cod72a] E. F. Codd, Further Normalization of the Data Base Relational Model, *Data Base Systems*, Englewood Cliffs, New Jersey, 1972, pp. 33-64.

[Cod72b] E. F. Codd, Relational Completeness of Data Base Sublanguages, *Data Base Systems*, Englewood Cliffs, New Jersey, 1972, pp. 65-98.

[Cod79] E. F. Codd, Extending the Database Relational Model to Capture More Meaning, *ACM Transactions on Database Systems vol. 4, no. 4*, (December 1979), pp. 397-434.

[Cul78] Cullinane, *IDMS DML Programmer's Reference Guide*, Cullinane Corporation, Wellesley, Massachusetts, 1978.

[Dat81] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981. 3rd ed.

[Dat83] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983. vol. II.

[Dat86] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986. 4rd ed..

[DoD81] United States DoD, *The Programming Language Ada: Reference Manual*, Springer-Verlag, New York, New York, 1981.

[Eps77] R. Epstein, A Tutorial on Ingres, Memorandum No. ERL-M77-25 (revised), University of California Berkeley, Berkeley, California, December, 1977.

[Fag77] R. Fagin, Multivalued Dependencies and a New Normal Form for Relational Databases, *ACM Transactions on Database Systems vol. 2, no. 3*, (September 1977), pp. 262-278.

[Fag79] R. Fagin, Normal Forms and Relational Database Operators, *ACM SIGMOD International Conference on Management of Data*, 1979, pp. 153-160.

[Fag81] R. Fagin, A Normal Form for Relational Databases that is Based on Domains and Keys, *ACM Transactions on Database Systems vol. 6, no. 3*, (September 1981), pp. 387-415.

[Ger83] V. Gertsberg, Example of the Semilattice Database, Unpublished, Department of Computing Science, University of Alberta, Edmonton, Alberta, 1983.

[Gre83] M. Green et. al., Experience with a Graphical Data Base System, *Graphics Interface 83*, May 1983, pp. 257-270.

[HOT76] P. Hall, J. Owlett and S. Todd, Relations and Entities, *Modelling in Data Base Management Systems*, Amsterdam, 1976, pp. 201-220.

[HaM81] M. Hammer and D. McLeod, Data Description with SDM: A Semantic Database Model, *ACM Transactions on Database Systems vol. 6, no. 3*, (September 1981), pp. 351-386.

[Har84] M. Hardwick, Extending the Relational Database Model for Design Applications, *Proceedings 21st Design Automation Conference*, 1984, pp. 110-116.

[Hay83] M. N. Haynie, Tutorial: The Relational Data Model for Design Automation, *Proceedings 20th Design Automation Conference*, 1983, pp. 599-607.

[Her83] H. Hernandez, Relational View Support using the Semilattice Model, Unpublished, Department of Computing Science, University of Alberta, Edmonton, Alberta, September 1983.

[HNC84] L. Hollaar, B. Nelson and T. Carter, The Structure and Operation of a Relational Database System in a Cell-Oriented Integrated Circuit Design System, *Proceedings 21st Design Automation Conference*, 1984, pp. 117-125.

[HuY84] R. Hull and C. K. Yap, The Format Model: A Theory of Database Organization, *Journal of the ACM vol. 31, no. 3*, (July 1984), pp. 518-537.

[IBM78] IBM, Information Management System/Virtual Storage General Information Manual, IBM Form No. GH20-1260, IBM, White Plains, New York, 1978.

[IBM80] IBM, Fast Path Feature Description and Design Guide, IBM Form No. G320-5775, IBM, White Plains, New York, 1980.

[Jac82] B. E. Jacobs, On Database Logic, *Journal of the ACM vol. 29, no. 2*, (April 1982), pp. 310-332.

[JSW83] H. R. Johnson, J. E. Schweitzer and E. R. Warkentine, A DBMS Facility for Handling Structured Engineering Entities, *Proceedings 1983 Data Base Week: Engineering Design Applications*, May 1983, pp. 3-12.

[KeW85] A. M. Keller and M. W. Wilkins, On the Use of an Extended Relational Model to Handle Changing Incomplete Information, *IEEE Transactions on Software Engineering vol. SE-11, no. 7*, (July 1985), pp. 620-633.

[Ken79] W. Kent, Limitations of Record-Based Information Models, *ACM Transactions on Database Systems vol. 4, no. 1*, (March 1979), pp. 107-131.

[Ken81] W. Kent, Consequences of Assuming a Universal Relation, *ACM Transactions on Database Systems vol. 6, no. 4*, (December 1981), pp. 539-556.

[Ken83a] W. Kent, A Simple Guide to Five Normal Forms in Relational Database Theory, *Communications of the ACM vol. 26, no. 2*, (February 1983), pp. 120-125.

[Ken83b] W. Kent, The Universal Relation Revisited, *ACM Transactions on Database Systems vol. 8, no. 4*, (December 1983), pp. 645-648.

[Kit84] H. Kitagawa et. al., Formgraphics: A Form-Based Graphics Architecture Providing a Database Workbench, *IEEE Computer Graphics and Applications vol. 4, no. 6*, (June 1984), pp. 38-56.

[Kor86] H. F. Korth, Extending the Scope of Relational Languages, *IEEE Software vol. 3, no. 1*, (January 1986), pp. 19-28.

[KuS82] S. M. Kuck and Y. Sagiv, A Universal Relation Database System Implemented via the Network Model, *Proceedings Symposium on Principles of Database Systems*, 1982, pp. 147-157.

[KuS84] S. M. Kuck and Y. Sagiv, Links in Relational Databases, *Proceedings CIPS Session 84*, 1984, pp. 131-137.

[KuV85] G. M. Kuper and M. Y. Vardi, On the Expressive Power of the Logical Data Model: Preliminary Report, *ACM SIGMOD International Conference on Management of Data*, 1985, pp. 180-187.

[LeF83] Y. C. Lee and K. S. Fu, Integration of Solid Modeling and Database Management for CAD/CAM, *Proceedings 20th Design Automation Conference*, 1983, pp. 367-373.

[Lei79] Y. E. Lein, Multivalued Dependencies with Null Values in Relational Databases, *Proceedings 1979 International Conference on Very Large Data Bases*, 1979, pp. 61-68.

[Lip81] W. Lipski, On Databases with Incomplete Information, *Journal of the ACM vol. 28, no. 1*, (January 1981), pp. 41-70.

[Lor82] R. Lorie, Issues in Databases for Design Applications, *File Structures and Data Bases for CAD*, Amsterdam, 1982, pp. 213-222.

[LoP83] R. Lorie and W. Plouffe, Complex Objects and Their Use in Design Transactions, *Proceedings 1983 Data Base Week: Engineering Design Applications*, May 1983, pp. 115-121.

[Lyn85] T. J. Lynch, *Data Compression: Techniques and Applications*, Lifetime Learning Publications, Belmont, California, 1985.

[MRI78] MRI, *System 2000 Reference Manual*, MRI Systems Corporation, Austin, Texas, 1978.

[Mai83] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, 1983.

[McG77] W. C. McGee, The IMS/VS System, *IBM Systems Journal vol. 16, no. 2*, (1977), pp. 84-168. Parts I-V.

[McL85] P. McLellan, Efficient Data Management for VLSI Design, *Proceedings 22nd Design Automation Conference*, 1985, pp. 652-657.

[Mer84] T. H. Merrett, *Relational Informational Systems*, Reston Publishing Company, Reston, Virginia, 1984.

[MMC76] A. S. Michaels, B. Miltman and C. R. Carlson, A Comparison of Relational and CODASYL Approaches to Data-Base Management, *ACM Computing Surveys vol. 8, no. 1*, (March 1976), pp. 125-151.

[Par82] J. Pardaens, A Universal Formalism to Express Decompositions, Functional Dependencies, and Other Constraints in a Relational Database, *Theoretical Computer Science vol. 19, no. 2*, (August 1982), pp. 161-187.

[Ris79] J. Rissanen, Theory of Joins for Relational Databases - A Tutorial Survey, *Proceedings Seventh Symposium on Mathematical Foundations of Computer Science vol. 26*, (1979), pp. 537-551, Springer-Verlag.

[Rob81] J. T. Robinson, The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes, *ACM SIGMOD International Conference on Management of Data*, 1981, pp. 10-18.

[SaR83] R. Sacks-Davis and K. Ramamohanarao, A Two Level Superimposed Coding Scheme for Partial Match Retrievals, *Information Systems vol. 8, no. 4*, (1983), pp. 273-280.

[Sac85] R. Sacks-Davis, Performance of a Multi-key Access Method Based on Descriptors and Superimposed Coding Techniques, *Information Systems vol. 10, no. 4*, (1985), pp. 391-403.

[SaU80] F. Sadri and J. D. Ullman, The Interaction Between Functional Dependencies and Template Dependencies, *ACM SIGMOD International Conference on Management of Data*, 1980, pp. 45-51.

[Sam76] J. R. Sampson, *Adaptive Information Processing*, Springer-Verlag, New York, New York, 1976.

[ScP82] H. J. Schek and P. Pistor, Data Structures for an Integrated Data Base Management and Information Retrieval System, *Proceedings 1982 International Conference on Very Large Data Bases*, September 1982, pp. 197-207.

[ScM85] E. Schell and M. R. Mercer, CADTOOLS: A CAD Algorithm Development System, *Proceedings 22nd Design Automation Conference*, 1985, pp. 658-666.

[Sch77] H. A. Schmid, An Analysis of Some Constructs for Conceptual Models, *Architecture and Models in Data Base Management Systems*, Amsterdam, 1977, pp. 119-148.

[Sci79] E. Sciore, Improving Semantic Specification in the Database Relational Model, *ACM SIGMOD International Conference on Management of Data*, 1979, pp. 170-178.

[Shi81] D. Shipman, The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems vol. 6, no. 1*, (March 1981), pp. 140-173.

[ShW85] A. Shoshami and H. K. T. Wong, Statistical and Scientific Database Issues, *IEEE Transactions on Software Engineering vol. SE-11, no. 10*, (October 1985), pp. 1040-1047.

[Sin85] A. Singh, Access Methods for a Semilattice Database Management System, Master's Thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Fall 1985.

[Soc85] G. H. Sockut, A Framework for Logical Level Changes Within Database Systems, *Computer vol. 18, no. 5*, (May 1985), pp. 9-27.

[Sof80] Relational Software, *ORACLE Introduction*, Relational Software Incorporated, Menlo Park, California, 1980.

[Spo84] D. L. Spooner, Database Support for Interactive Computer Graphics, *ACM SIGMOD International Conference on Management of Data*, 1984, pp. 90-99.

[SWK76] M. Stonebraker, E. Wong, P. Kreps and G. Held, The Design and Implementation of INGRES, *ACM Transactions on Database Systems vol. 1, no. 3*, (September 1976), pp. 189-222.

[Sto80] M. Stonebraker, Retrospection on a Database System, *ACM Transactions on Database Systems vol. 5, no. 2*, (June 1980), pp. 225-240.

[SRG83] M. Stonebraker, B. Rubenstein and A. Guttman, Application of Abstract Data Types and Abstract Indices to CAD Databases, *Proceedings 1983 Data Base Week: Engineering Design Applications*, May 1983, pp. 107-114.

[Sto83] M. Stonebraker et. al., Performance Enhancements to a Relational Database System, *ACM Transactions on Database Systems vol. 8, no. 2*, (June 1983), pp. 167-185.

[Sto86] M. Stonebraker et. al., Document Processing in a Relational Database System, *The Ingres Papers: Anatomy of a Relational Database System*, Reading, Massachusetts, 1986, pp. 357-375.

[Su86] S. Y. W. Su, Modeling Integrated Manufacturing Data with SAM†, *Computer vol. 19, no. 1*, (January 1986), pp. 34-49.

[UMO82] S. Ulfsby, S. Meen and J. Oian, Tornado: A Data-Base Management System for Graphics Applications, *IEEE Computer Graphics and Applications vol. 2, no. 3*, (May 1982), pp. 71-79.

[Ull82] J. D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, Maryland, 1982. 2nd ed..

[Ull83] J. D. Ullman, On Kent's Consequences of Assuming a Universal Relation, *ACM Transactions on Database Systems vol. 8, no. 4*, (December 1983), pp. 637-643.

[Vas79] Y. Vassiliou, Null Values in Database Management - A Denotational Semantics Approach, *ACM SIGMOD International Conference on Management of Data*, 1979, pp. 162-169.

[Vas80] Y. Vassiliou, Functional Dependencies and Incomplete Information, *Proceedings 1980 International Conference on Very Large Data Bases*, 1980, pp. 260-269.

[WeY84] D. L. Weller and B. W. York, A Relational Representation of an Abstract Type System, *IEEE Transactions on Software Engineering vol. SE-10, no. 3*, (May 1984), pp. 303-309.

[Wie77] G. Wiederhold, *Database Design*, McGraw-Hill Book Company, New York, New York, 1977.

[Wil84] R. B. Wilmot, Foreign Keys Decrease Adaptability of Database Designs, *Communications of the ACM vol. 27, no. 12*, (December 1984), pp. 1237-1243.

[WoS86] J. Woodfill and M. Stonebraker, An Implementation of Hypothetical Relations, *The Ingres Papers: Anatomy of a Relational Database System*, Reading, Massachusetts, 1986, pp. 334-356.

[Woo82] J. Woodfill et. al., *Ingres Version 7 Reference' Manual*, University of California Berkeley, Berkeley, California, March 1982.

[YaK82] K. Yamaguchi and T. L. Kunii, PICCOLO Logic for a Picture Database Computer and its Implementation, *IEEE Transactions on Computer Systems vol. C-31, no. 10*, (October 1982), pp. 983-996.

Appendix A1

Conceptual Language BNF

A1.1. Conceptual Language Specification

The following specification describes the Conceptual Language with a variant of BNF similar to that used in the Ada Reference Manual [DoD81]. Optional items are enclosed in square brackets "[]", alternative items are separated by a vertical bar "|", and items which can be repeated zero or more times are enclosed in square brackets followed by an asterisk "[]*". The vertical bar stands for itself when appearing immediately following a left bracket "[|". Nonterminals begin with a capital letter while terminal symbols begin in lower case.

1.1.1. Data Definition

Domain ::=	Simple_domain | Tuple_domain | {Simple_domain} | {Tuple_domain} | <Simple_domain> | <Tuple_domain> | Dis_union_domain
Complex_domain ::=	(Attribute_domain [,Attribute_domain]*) [,Constraint] [,Meaning]
Attribute_domain ::=	Attribute_name : Domain
Dis_union_domain ::=	(Tag : Domain [| Tag : Domain]*)
Simple_domain ::=	integer | real | boolean | string
Tuple_domain ::=	Name
Attribute_name ::=	Name
Tag ::=	Name
Constraint ::=	constraint: "Qualification"
Meaning ::=	meaning: "String"
Name ::=	Letter [Letter | Digit | Underscore]*
Letter ::=	a | b | ... | z | A | B | ... | Z
Digit ::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Underscore ::=	_

```
Domain_value ::=      Simple_value | Complex_value |
                      {[Simple_value [,Simple_value]*]} |
                      {[Complex_value [,Complex_value]*]} |
                      <[Simple_value [,Simple_value]*]> |
                      <[Complex_value [,Complex_value]*]> |
                      Dis_union_value

Complex_value ::=     (Attribute_value [,Attribute_value]*)

Attribute_value ::=   [Attribute_name :=] Domain_value

Dis_union_value ::=   (Tag , Domain_value )

Simple_value ::=      Integer | Real | Boolean | String

Integer ::=           [+|-] Number

Real ::=              [+|-] Number.Number

Number ::=            Digit [Digit]*

Boolean ::=           true | false

String ::=            Letter | Digit | Underscore |
                      ! | @ | # | % | ^ | & | * | (| | ) | - | + | = |
                      { | } | [ | ] | ; | : | , | < | . | > | / | ? | ~ | ` | '
```

1.1.2. Data Manipulation

Procedural ::=	Declaration | Update
Declaration ::=	Type_name = Domain | Type_name = Complex_domain | Variable_name : Type_name
Type_name ::=	Name
Var_declaration ::=	Name
Update ::=	Insert | Delete | Modify
Insert ::=	insert Variable into Variable_name
Delete ::=	delete Variable from Variable_name
Modify ::=	modify Variable with Modification [where Qualification]
Variable ::=	Variable_name | Domain_value
Qualification ::=	Qualifier [Connective Qualifier]*
Qualifier ::=	[Unary] Argument Condition Argument
Argument ::=	Qualified_name | Domain_value
Connective ::=	and | or
Unary ::=	not
Condition ::=	= | != | < | > | <= | >= | in
Modification ::=	Modifier [Connective Modifier]*
Modifier ::=	Argument Operator Argument
Operator ::=	+ | - | * | /

Statement ::=	Operation | Print
Qualified_name ::=	Variable_name[.Attribute_name] [;Tag]
Operation ::=	Union | Difference | Product | Project | Select
Union ::=	Variable_name := union(Variable,Variable)
Difference ::=	Variable_name := difference(Variable,Variable)
Product ::=	Variable_name := product(Variable,Variable)
Project ::=	Variable_name := Variable_name
Select ::=	Variable_name := Variable_name where Qualification
Print ::=	print Variable_name

Appendix A2

Source Code Listing

2.1. Conceptual Level Software Source Code

This appendix contains the complete source code for the conceptual level software of the Semilattice Database Management System. This code may be run standalone or with the $C-B^+-tree$ software by omitting the stub.c interface.

```
#define MAX_RESERVED    24                      /* reserved words */
#define MAX_SPECIAL     4                       /* special symbols */
#define MAX_BUFFER      256                     /* input buffer size */
#define MAX_PARSE       128                     /* parse buffer size */
#define NULL_CHAR       127                     /* last buffer - always empty */
#define NULL_STR        ""                      /* null string */
#define MAX_IDENTIFIER  16                      /* unique identifier characters */
#define MAX_STRING      32                      /* maximum string length */
#define MAX_RECORDS     32                      /* largest record buffer */
#define MAX_SOURCE      2                       /* parser source pointers */
#define MAX_QUAL        16                      /* parser qualification pointers */

#define FN_FAIL         0                       /* function failure return code */
#define FN_OK           1                       /* function success return code */
#define FN_END          2                       /* function complete */
#define FN_MORE         3                       /* function more data */

#define PUNCT           1                       /* punctuation */
#define SPEC            2                       /* special character */
#define RESRV           3                       /* reserved word */
#define NUMB            4                       /* number */
#define ALPH            5                       /* identifier */

#define T_INTEGER       0                       /* type integer */
#define T_REAL          1                       /* type real */
#define T_BOOLEAN       2                       /* type boolean */
#define T_STRING        3                       /* type string */
#define T_SET           4                       /* type set */
#define T_SEQUENCE      5                       /* type sequence */
#define T_TUPLE         6                       /* type tuple */
#define T_UNION         7                       /* type disjoint union */
#define T_TYPE          8                       /* type defined type */
#define T_ERROR         255                     /* error condition */

#define MASK_NESTED_TUPLE 0x01                  /* nested tuple flag for coding */
#define MASK_NESTED_SET   0x02                  /* nested set flag for coding */


union   sl_type                                 /* variant data record */
    {
    int     ival;
    float   fval;
    char    sval[ MAX_STRING ];
    };

struct  st_rec                                  /* symbol table record */
    {
    char    name[ MAX_IDENTIFIER+1 ];
    int     kind;
    struct  st_rec  *base_ptr;
    struct  st_rec  *next_ptr;
    union   sl_type val;
    };

#define NIL_ST_REC   ((struct st_rec *) 0)
```



```
/* B-tree interface constants and types */

#define MAX_INTER_DATA 128                      /* maximum interface data buffer */

/* Find function flags */

#define MASK_FIND_LEFT 0x01                     /* find leftmost tree record */
#define MASK_FIND_NEXT 0x02                     /* find next record given key */
#define MASK_FIND_KEY  0x04                     /* find record data part given key */

/* Insert function flags */

#define MASK_INS_REPLACE 0x01                   /* insert replace record flag */


struct  data_rec                                /* B-tree data record */
    {
    int     flags;
    int     length;
    char    data[ MAX_INTER_DATA ];
    };
```

Jan 20 20:52 1987 token.h Page 1

```
# define NE 257
# define LE 258
# define GE 259
# define ASSIGN 260
# define INTEGER 261
# define REAL 262
# define BOOLEAN 263
# define STRING 264
# define CONSTRAINT 265
# define MEANING 266
# define INSERT 267
# define INTO 268
# define WHERE 269
# define DELETE 270
# define FROM 271
# define MODIFY 272
# define WITH 273
# define UNION 274
# define DIFFERENCE 275
# define PRODUCT 276
# define PRINT 277
# define AND 278
# define OR 279
# define NOT 280
# define TRUE 281
# define FALSE 282
# define SUB 283
# define QUIT 284
# define NUMBER 285
# define ALPHA 286
# define TYPE_DECL 287
# define VAR_DECL 288
# define PROJECT 289
# define SELECT 290
# define PRECOR 291
# define PRECAND 292
# define PRECEXP 293
# define PRECUNARY 294
```

Jan 20 20:56 1987 parser.y Page 1

```
%{
#include    <stdio.h>
#include    <strings.h>
#include    <ctype.h>
#include    "defines.h"
%}

%start  Semilattice

%token  NE LE GE ASSIGN
%token  INTEGER REAL BOOLEAN STRING CONSTRAINT MEANING
%token  INSERT INTO WHERE DELETE FROM MODIFY
%token  WITH UNION DIFFERENCE PRODUCT PRINT
%token  AND OR NOT TRUE FALSE SUB QUIT NUMBER ALPHA
%token  TYPE_DECL VAR_DECL PROJECT SELECT

%right  '='
%left   PRECOR
%left   PRECAND
%left   '+' '-'
%left   '*' '/' '%'
%left   PRECEXP
%left   PRECUNARY

%% /* start of rules */

Semilattice     : /* empty */
                | Semilattice Procedural
                    { Print_prompt( ); }
                | Semilattice Statement
                    { Print_prompt( ); }
                | Semilattice System
                    { Print_prompt( ); }
                ;

Domain          : Simple_domain
                | Tuple_domain
                | '[' Simple_domain ']'
                | '[' Tuple_domain ']'
                | '<' Simple_domain '>'
                | '<' Tuple_domain '>'
                | Dis_union_domain
                ;

Complex_domain  : '(' Att_dom ')' Tail
                ;

Tail            : ',' Constraint ',' Meaning
                | ',' Constraint
                | ',' Meaning
                | /* empty */
                ;

Att_dom         : Att_dom ',' Attribute_domain
                | Attribute_domain
                ;
```

```
Attribute_domain : Attribute_name ':' Domain
                 ;

Dis_union_domain : '(' Tag_dom ')'
                 ;

Tag_dom          : Tag_dom '|' Tag ':' Domain
                 | Tag ':' Domain
                 ;

Simple_domain    : INTEGER
                 | REAL
                 | BOOLEAN
                 | STRING
                 ;

Tuple_domain     : Name
                 ;

Attribute_name   : Name
                 ;

Tag              : Name
                 ;

Constraint       : CONSTRAINT ':' '"' Qualification '"'
                 ;

Meaning          : MEANING ':' '"' Text '"'
                 ;

Text             : Text All
                 | All
                 ;

All              : NUMBER | ALPHA
                 | '!' | '@' | '#' | '$' | '%' | '^' | '&' | '*' | ')'
                 | '(' | '-' | '_' | '=' | '+' | '~' | '`' | ';' | ':'
                 | '\'' | '[' | ']' | '{' | '}' | ',' | '.' | '<' | '>'
                 | '/' | '?'
                 ;

Name             : Name ALPHA
                 | ALPHA
                 ;

Domain_value     : Simple_value
                 | Complex_value
                 '(' ')'
                 '(' Simple_values ')'
                 '(' Complex_values ')'
                 '<' '>'
                 '<' Simple_values '>'
                 '<' Complex_values '>'
                 | Dis_union_value
```



```
                 ;

Complex_value    : '(' Att_val ')'
                 ;

Att_val          : Att_val ',' Attribute_value
                 | Attribute_value
                 ;

Attribute_value  : Attribute_name ASSIGN Domain_value
                 | Domain_value
                 ;

Dis_union_value  : '(' Tag ':' Domain_value ')'
                 ;

Simple_values    : Simple_values ',' Simple_value
                 | Simple_value
                 ;

Simple_value     : Integer
                 | Real
                 | Boolean
                 | String
                 ;

Complex_values   : Complex_values ',' Complex_value
                 | Complex_value
                 ;

Integer          : '+' Number
                 | '-' Number
                 | Number
                 ;

Real             : '+' Real_num
                 | '-' Real_num
                 | Real_num
                 ;

Real_num         : Number '.' Number
                 ;

Number           : Number NUMBER
                 | NUMBER
                 ;

Boolean          : TRUE
                 | FALSE
                 ;

String           : '"' Text '"'
                 | '"' '"'
                 ;

Procedural       : Declaration '\n'
```

```
                | Tuple_update '\n'
                ;

Declaration     : Type_declaration
                    { Intermediate( TYPE_DECL ); }
                | Var_declaration
                    { Intermediate( VAR_DECL ); }
                ;

Type_declaration : Type_name '=' Domain
                | Type_name '=' Complex_domain
                ;

Var_declaration : Variable_name ':' Type_name
                ;

Type_name       : Name
                ;

Variable_name   : Name
                ;

Tuple_update    : Insert
                    { Intermediate( INSERT ); }
                | Delete
                    { Intermediate( DELETE ); }
                | Modify
                    { Intermediate( MODIFY ); }
                ;

Insert          : INSERT Variable INTO Variable_name
                ;

Delete          : DELETE Variable FROM Variable_name
                ;

Modify          : MODIFY Variable_name WITH Modification WHERE Qualification
                | MODIFY Variable_name WITH Modification
                ;

Variable        : Variable_name
                | Domain_value
                ;

Qualification   : Qualification Connective Qualifier
                | Qualifier
                ;

Modification    : Modification Connective Modifier
                | Modifier
                ;

Qualifier       : Argument Condition Argument
                | Unary Argument Condition Argument
                ;
```



```
Modifier        : Argument Operator Argument
                ;

Argument        : Qualified_name
                | Domain_value
                ;

Condition       : '='
                | NE
                | '<'
                | '>'
                | LE
                | GE
                | SUB
                ;

Connective      : AND
                | OR
                ;

Unary           : NOT
                ;

Operator        : '+'
                | '-'
                | '*'
                | '/'
                ;

Statement       : Operation '\n'
                | Print '\n'
                ;

Qualified_name  : Att_name ';' Tag
                | Att_name
                ;

Att_name        : Att_name '.' Attribute_name
                | Variable_name
                ;

Operation       : Union
                    { Intermediate( UNION ); }
                | Difference
                    { Intermediate( DIFFERENCE ); }
                | Product
                    { Intermediate( PRODUCT ); }
                | Project
                    { Intermediate( PROJECT ); }
                | Select
                    { Intermediate( SELECT ); }
                ;

Union           : Variable_name ASSIGN UNION '(' Vars ')'
                ;
```

```
Vars            : Variable ',' Variable
                ;

Difference      : Variable_name ASSIGN DIFFERENCE '(' Vars ')'
                ;

Product         : Variable_name ASSIGN PRODUCT '(' Vars ')'
                ;

Project         : Variable_name ASSIGN Variable_name
                ;

Select          : Variable_name ASSIGN Variable_name WHERE Qualification
                ;

Print           : PRINT Variable_name
                  { Intermediate( PRINT ); }
                ;

System          : QUIT '\n'
                  { fprintf( stderr , "\n" ); exit( 0 ); }
                | '\n' /* empty */
                ;

%% /* start of code */

static  char *reserved[ ] =
    {
    "integer",      "real",         "boolean",
    "string",       "constraint",   "meaning",
    "insert",       "into",         "where",
    "delete",       "from",         "modify",
    "with",         "union",        "difference",
    "product",      "print",        "and",
    "or",           "not",          "true",
    "false",        "in",           "quit"
    };

static  char *special[ ] =
    {
    "!=" , "<=" , ">=" , ":="
    };

int     c , lookahead;                          /* current and next character */
int     err , last_token;                       /* error buffer pointers */
char    err_buf[ MAX_BUFFER ];                  /* error buffer */
FILE    *fpp;                                   /* input file pointer */

int     parse_buf[ MAX_PARSE ];                 /* input parser buffer */
int     parse_int[ MAX_PARSE ];         /* enumerated type and number buffer */
char    parse_chr[ MAX_PARSE ][ MAX_IDENTIFIER+1 ];     /* identifier buffer */
int     parse_buf_ptr,scan;                     /* buffer pointers */

int     operator , target;          /* command operator and target variable */
int     source[ MAX_SOURCE ];               /* command source variables index */
```



```
struct  st_rec  *var_head;                      /* variable list head */
struct  st_rec  *type_head;                     /* type list head */

extern void Error( );
extern void Interpretter( );

void List_vars( h )
    struct  st_rec  *h;
    {

    while ( h != NULL )
        {
        fprintf( stderr , "%-18s: %s\n" , h->name , h->base_ptr->name );
        h = h->next_ptr;
        }
    } /* List_vars */

char Get_char( )
    {
    char    ch;                                 /* input character */

    if ( fpp == NULL )
        return( getchar( ) );
    else
        {
        if ( ( ch = getc( fpp ) ) == EOF )
            {
            fclose( fpp );
            fpp = NULL;
            ch = '\n';
            fprintf( stderr , "\nload done\n" );
            List_vars( var_head );
            }
        return( ch );
        }
    } /* Get_char */

void Print_prompt( )
    {
    if ( fpp == NULL )
        fprintf( stderr , "\nSL > " );
    else
        fprintf( stderr , "." );
    } /* Print_prompt */

void Fill_chr_buf( s )
    char    *s;                                 /* character string */
    {
```

```
    parse_buf[ parse_buf_ptr ] = ALPH;
    (void) strcpy( parse_chr[parse_buf_ptr] , s );
    parse_chr[parse_buf_ptr++][MAX_IDENTIFIER] = '\0';
    } /* Fill_chr_buf */


void Fill_int_buf( kind , i )
    int     kind;                       /* kind of token */
    int     i;                          /* integer value */
    {
#if DEBUG_PARSE
    int     b;                          /* index */
#endif

    parse_buf[ parse_buf_ptr ] = kind;
    parse_int[ parse_buf_ptr++ ] = i;

    if (( c == '\n' ) && ( lookahead == 0 ))
        {
#if DEBUG_PARSE
        b = 0;

        while ( b != parse_buf_ptr )
            {
            fprintf( stderr , "kind  = %d  " , parse_buf[ b ] );
            if ( parse_buf[ b ] == ALPH )
                fprintf( stderr , "alph  = %s\n" , parse_chr[ b++ ] );
            else if ( parse_buf[ b ] == PUNCT )
                fprintf( stderr , "punct = %c\n" , parse_int[ b++ ] );
            else
                fprintf( stderr , "numb  = %d\n" , parse_int[ b++ ] );
            }
#endif
        parse_buf_ptr = 0;
        }
    } /* Fill_int_buf */


int yylex( )                            /* lexical analyzer */
    {
    int     p;                          /* last characater type flag */
    int     i , j , k;                  /* counters */
    char    token[ MAX_BUFFER ];        /* input strings */

    /* reset error buffer pointers */

    if ( c == '\n' )
        {
        err = 0;
        last_token = 0;
        }
    else
        last_token = err;

    /* check for lookahead character */
```



```
    if ( lookahead == 1 )
        if ( ispunct ( c ) && c != ';' && c != '(' && c != ')' && c != ',' )
            {
            lookahead = 0;
#if DEBUG_PARSE
            fprintf( stderr , "punct = %c\n" , c );
#endif
            Fill_int_buf( PUNCT , c );
            return( c );
            }
        else
            lookahead = 0;
    else

        /* remove whitespace */

        while (( err_buf[err++] = c = Get_char( )) == ' ' || c == '\t' );

    /* endif */

    token[ 0 ] = c;
    p = ispunct( c );
    i = 0;

    /* read complete token */

    if ( c != '\n' )
        while (( err_buf[err++] = c = Get_char( )) != ' ' && c != '\t' )
            {
            if ( ( p == 0 && ispunct( c ) && c != '_' ) ||
                 ( c == '\n' ) ||
                 ( p != 0 && c != '-' ))
                {
                /* end of token, save flag lookahead character */

                lookahead = 1;
                break;
                }

            token[ ++i ] = c;
            } /* end do */

    /* return character or token */

    if ( ( i == 0 ) && ( ispunct( token[ 0 ] ) ||
         token[ 0 ] == '\n' ) )
        {
#if DEBUG_PARSE
        fprintf( stderr , "punct = %c\n"  , token[ 0 ] ) ;
#endif
        Fill_int_buf( PUNCT , token[ 0 ] );
        return( token[ 0 ] );
        }

    token[ ++i ] = '\0';
```

```
    for ( j = 0 ; j < MAX_SPECIAL ; j++ )
        if ( strcmp( special[ j ] , token ) == 0 )
            {
#if DEBUG_PARSE
            fprintf( stderr , "spec  = %s\n" , special[ j ] );
#endif
            Fill_int_buf( SPEC , NE + j );
            return( NE + j );
            }

    for ( j = 0 ; j < MAX_RESERVED ; j++ )
        if ( strcmp( reserved[ j ] , token ) == 0 )
            {
#if DEBUG_PARSE
            fprintf( stderr , "resrv = %s\n" , reserved[ j ]);
#endif
            Fill_int_buf( RESRV , INTEGER + j );
            return( INTEGER + j );
            }

    yylval = 0;
    j = 0;

    /* reverse digits for real fraction part to preserve leading zeros */

    if ( ( parse_buf[ parse_buf_ptr-1 ] == PUNCT ) &&
         ( parse_int[ parse_buf_ptr-1 ] == '.' ) )
        {
        while ( ( j < i ) && isdigit( token[ j ] ) )
            j++;

        k = j;
        while ( k >-0 )
            yylval = yylval * 10 + ( token[ --k ] - '0' );
        }

    else
        while ( ( j < i ) && isdigit( token[ j ] ) )
            yylval = yylval * 10 + ( token[ j++ ] - '0' );

    if ( j == i )
        {
#if DEBUG_PARSE
        fprintf( stderr , "numb  = %d\n" , yylval );
#endif
        Fill_int_buf( NUMB , yylval );
        return( NUMBER );
        }

#if DEBUG_PARSE
    fprintf( stderr , "alpha = %s\n"  , token );
#endif
    Fill_chr_buf( token );
    return( ALPHA );
```



```
    } /* yylex */

void Intermediate( op )
    int     op;                                     /* operator */
    {
    int     i;                                      /* index */

    /* initialization */

    operator = op;
    target = source[ 0 ] = source[ 1 ] = 0;

    /* set target and source pointers depending on operation */

    switch( operator )
        {
        case TYPE_DECL:
        case VAR_DECL:
            source[ 0 ] = 2;
            break;

        case INSERT:
            source[ 0 ] = 1;
            while ( parse_buf[ target ] != RESRV ||
                    parse_int[ target ] != INTO ) target++;
            target++;
            break;

        case DELETE:
            if ( parse_buf[ 1 ] != RESRV )
                source[ 0 ] = 1;
            while ( parse_buf[ target ] != RESRV ||
                    parse_int[ target ] != FROM ) target++;
            target++;
            break;

        case MODIFY:
            i = 0;
            while ( i < MAX_PARSE )
                if ( parse_buf[ i ] == RESRV && parse_int[ i ] == WHERE )
                    {
                    source[ 1 ] = i + 1;
                    break;
                    }
                else
                    i++;

            target = 1;
            while ( parse_buf[ source[0] ] != RESRV ||
                    parse_int[ source[0] ] != WITH ) source[0]++;
            source[ 0 ]++;
            break;

        case UNION:
        case DIFFERENCE:
```

```
        case PRODUCT:
            source[ 0 ] = source[ 1 ] = 4;
            i = 0;

            /* scan for comma not in a value */

            while ( source[1] < MAX_PARSE )
                {
                if ( parse_buf[ source[1] ] == PUNCT )
                    {
                    if ( ( parse_int[ source[1] ] == ',' ) && ( i == 0 ) )
                        break;

                    else if ( ( parse_int[ source[1] ] == '(' ) ||
                              ( parse_int[ source[1] ] == '<' ) ||
                              ( parse_int[ source[1] ] == '{' ) )
                        /* nest */
                        i++;

                    else if ( ( parse_int[ source[1] ] == ')' ) ||
                              ( parse_int[ source[1] ] == '>' ) ||
                              ( parse_int[ source[1] ] == '}' ) )
                        /* unnest */
                        i--;

                    }
                source[ 1 ]++;
                }
            source[ 1 ]++;
            break;

        case PROJECT:
            source[ 0 ] = 2;
            break;

        case SELECT:
            source[ 0 ] = 2;
            while ( parse_buf[ source[1] ] != RESRV ||
                    parse_int[ source[1] ] != WHERE ) source[1]++;
            source[ 1 ]++;
            break;

        case PRINT:
            target = 1;
            break;

        } /* switch */

#if DEBUG_PARSE
    fprintf( stderr, "t=%d  op=%d  s1=%d  s2=%d\n",
            target,operator,source[0],source[1]);
#endif
    Interpretter( );

    } /* Intermediate */
```



```
void yyerror( s )
    char    *s;                                     /* error message
    {

    fprintf( stderr , "%s\n" , s );

    /* read remaining input */

    if ( ( err_buf[ --err ] != '\n' ) && c != '\n' )
        while ( ( err_buf[ ++err ] = Get_char( ) ) != '\n' );

    for ( err = 0 ; err_buf[ err ] != '\n' ; err++ );

    err_buf[ err ] = '\0';

    /* print error indicator */

    fprintf( stderr , "%s\n" , err_buf );

    while ( last_token-- != 0 )
        fprintf( stderr , " " );

    fprintf( stderr , "|\n\nSL > " );
    err = 0;
    last_token = 0;
    lookahead = 0;
    parse_buf_ptr = 0;

    /* tell parser error is OK */

    yyerrok;
    yyclearin;

    } /* yyerror */


void Init_sl( )
    {

    /* clear screen and display start-up header */

    system( "clear" );
    fprintf( stderr , "\nSemilattice Database\n\n" );
    fprintf( stderr , "Loading database\n" );

    lookahead = 0;
    c = '\n';

    /* initialize symbol table */

    parse_chr[ NULL_CHAR ][ 0 ] = '\0';
    var_head = type_head = NULL;

    /* load database */
```

```
    if ( ( fpp = fopen( ".sl_init" , "r" ) ) == NULL )
        {
        Error( "No load file found" );
        fprintf( stderr , "\nSL > ");
        }

    } /* Init_sl */


main( )
    {
    Init_sl( );
    for (;;)
        yyparse( );
    } /* main */
```



```
#include    <stdio.h>
#include    <strings.h>
#include    <ctype.h>
#include    "defines.h"
#include    "token.h"

extern  int     parse_buf[ MAX_PARSE ];             /* input parser buffer */
extern  int     parse_int[ MAX_PARSE ]; /* enumerated type and number buffer */
extern  char    parse_chr[ MAX_PARSE ][ MAX_IDENTIFIER+1 ];  /* ident buffer */
extern  int     parse_buf_ptr,scan;                 /* buffer pointers */

extern  int     operator , target;   /* command operator and target variable */
extern  int     source[ MAX_SOURCE ];       /* command source variables index */


extern  struct  st_rec  *var_head;                  /* variable list head */
extern  struct  st_rec  *type_head;                   /* type list head */

extern char *calloc( );
extern char *Print( );
extern void Sl_insert( );
extern void Sl_delete( );
extern void Union( );
extern void Difference( );
extern void Pproject( );
extern void Product( );
extern void Modify( );
extern void Select( );


void Error( s )
    char    *s;                                     /* error message */
    {
    fprintf( stderr , "\ERROR: %s\n" , s );
    } /* Error */


struct st_rec *Find_st_name( h , s )
    struct  st_rec  *h;                             /* list head */
    int     s;                                      /* identifier name */
    {

    /* scan list for name */

    while ( h != NIL_ST_REC )
        {
        if ( strcmp( h->name , parse_chr[ s ] ) == 0 )
            /* found the name */

            return( h );

        h = h->next_ptr;
        }
```

```
    return( NIL_ST_REC );
    } /* Find_st_name */


struct st_rec *Make_st_rec( s , k , b_ptr , n_ptr )
    char    *s;                                     /* identifier name */
    int     k;                                      /* kind of record */
    struct  st_rec  *b_ptr;                         /* base pointer */
    struct  st_rec  *n_ptr;                         /* next pointer */
    {
    struct  st_rec  *p;                             /* new record pointer */

    /* allocate and cast a record */

    p = (struct st_rec *) calloc( 1 , sizeof(struct st_rec) );

    /* fill record fields */

    (void) strcpy( p->name , s );
    p->kind = k;
    p->base_ptr = b_ptr;
    p->next_ptr = n_ptr;

    return( p );
    } /* Make_st_rec */


struct st_rec *Build_st_type( tn )
    int     tn;                                     /* type name */
    {
    struct  st_rec  *rc;                            /* return pointer */
    struct  st_rec  *t_ptr;                         /* type pointer */
    struct  st_rec  *head,*last;                    /* list pointers */
    int     kind;                                   /* kind of record */

    rc = NIL_ST_REC;

    /* check for standard type declarations */

    if ( parse_buf[ scan ] == RESRV )
        {
        if ( parse_int[ scan ] == INTEGER )
            kind = T_INTEGER;
        else if ( parse_int[ scan ] == REAL )
            kind = T_REAL;
        else if ( parse_int[ scan ] == BOOLEAN )
            kind = T_BOOLEAN;
        else
            kind = T_STRING;

        rc = Make_st_rec( parse_chr[tn] , kind , NIL_ST_REC , NIL_ST_REC );
        scan++;
        }

    else if ( parse_buf[ scan ] == PUNCT )
        {
```



```
        if ( parse_int[ scan ] == '{' )
            {
            scan++;
            rc = Build_st_type( NULL_CHAR );
            if ( rc != NIL_ST_REC )
                rc = Make_st_rec( parse_chr[tn] , T_SET , rc , NIL_ST_REC );
            scan++;
            }
        else if ( parse_int[ scan ] == '<' )
            {
            scan++;
            rc = Build_st_type( NULL_CHAR );
            if ( rc != NIL_ST_REC )
                rc = Make_st_rec( parse_chr[tn] , T_SEQUENCE , rc , NIL_ST_REC);
            scan++;
            }

        else if ( parse_int[ scan ] == '(' )
            {
            if ( parse_int[ scan+2 ] == ':' )
                kind = T_TUPLE;
            else
                kind = T_UNION;

            head = last = NIL_ST_REC;

            /* make a list of all attributes or tags */

            while ( parse_int[ scan ] != ')' )
                {
                scan = scan + 3;
                rc = Build_st_type( scan-2 );

                if ( rc == NIL_ST_REC )
                    {
                    head = NIL_ST_REC;
                    break;
                    }
                else if ( last == NIL_ST_REC )
                    head = rc;
                else
                    last->next_ptr = rc;

                last = rc;
                }
            scan++;

            if ( head != NIL_ST_REC )
                {
                rc = Make_st_rec( parse_chr[tn] , kind , head , NIL_ST_REC );

                /* check for duplicate attributes/tags */

                while ( head != NIL_ST_REC )
                    {
                    last = head->next_ptr;
```

```
                while ( last != NIL_ST_REC )
                    {
                    if ( strcmp( head->name , last->name ) == 0 )
                        {
                        if ( kind == T_TUPLE )
                            Error( "Duplicate attribute name");
                        else
                            Error( "Duplicate tag name" );
                        rc = NIL_ST_REC;
                        break;
                        }
                    last = last->next_ptr;
                    }
                head = head->next_ptr;
                }
            }
        }

    else if ( parse_buf[ scan ] == ALPH )
        {
        if (( t_ptr = Find_st_name( type_head , scan )) == NIL_ST_REC )
            Error( "Undefined type" );
        else
            {
            rc = Make_st_rec( parse_chr[tn] , T_TYPE , t_ptr , NIL_ST_REC );
            scan++;
            }
        }

    else
       Error( "Build_st_type: var not defined" );

    return( rc );
    } /* Build_st_type */


struct st_rec *Build_st_value( )
    {
    struct  st_rec  *rc;                                /* return pointer */
    struct  st_rec  *head,*last;                        /* list pointers */
    int     kind;                                       /* kind of record */
    char    c;                                  /* terminating scheme character */

    rc = NIL_ST_REC;

    if ( parse_buf[ scan ] == PUNCT )
        if ( ( parse_int[ scan ] == '+' ) || ( parse_int[ scan ] == '-' ) )
            {
            /* skip the character, check for negation later */

            scan++;
            }

    if ( parse_buf[ scan ] == RESRV )
        {
```



```
        /* reserved word */

        rc = Make_st_rec( NULL_STR , T_BOOLEAN , NIL_ST_REC , NIL_ST_REC );
        rc->val.ival = FALSE - parse_int[ scan++ ];
        }

    else if ( parse_buf[ scan ] == NUMB )
        if ( ( parse_buf[ scan+1 ] == PUNCT ) &&
             ( parse_int[ scan+1 ] == '.' ) )
            {
            /* real number reverse fractional part */

            rc = Make_st_rec( NULL_STR , T_REAL , NIL_ST_REC , NIL_ST_REC );

            rc->val.fval = parse_int[ scan ];
            kind = 10;
            while ( parse_int[ scan+2 ] != 0 )
                {
                rc->val.fval += ( parse_int[ scan+2 ] % 10 ) / (float) kind;
                kind *= 10;
                parse_int[ scan+2 ] /= 10;
                } /* while */

            if ( ( parse_buf[ scan-1 ] == PUNCT ) &&
                 ( parse_int[ scan-1 ] == '-' ) &&
                 ( ( parse_buf[ scan-2 ] == PUNCT ) ||
                   ( parse_buf[ scan-2 ] == SPEC ) ) )
                rc->val.fval = 0 - rc->val.fval;

            scan = scan + 3;
            }
        else
            {
            /* integer number */

            rc = Make_st_rec( NULL_STR , T_INTEGER , NIL_ST_REC , NIL_ST_REC );
            rc->val.ival = parse_int[ scan ];

            if ( ( parse_buf[ scan-1 ] == PUNCT ) &&
                 ( parse_int[ scan-1 ] == '-' ) &&
                 ( ( parse_buf[ scan-2 ] == PUNCT ) ||
                   ( parse_buf[ scan-2 ] == SPEC ) ) )
                rc->val.ival = 0 - rc->val.ival;

            scan++;
            }

    else if ( parse_buf[ scan ] == PUNCT )
        {
        if ( parse_int[ scan ] == '"' )
            {
            /* string */

            rc = Make_st_rec( NULL_STR , T_STRING , NIL_ST_REC , NIL_ST_REC );
            scan++;
            rc->val.sval[0] = '\0';
```

```
    while ( ( parse_buf[ scan ] != PUNCT ) ||
            ( parse_int[ scan ] != '"' ) )
        {
        if ( parse_buf[ scan ] == NUMB )
            (void) sprintf( parse_chr[scan] , "%d" , parse_int[scan] );
        else if ( parse_buf[ scan ] == PUNCT )
            (void) sprintf( parse_chr[scan] , "%c" , parse_int[scan] );
        (void) strcat( rc->val.sval , parse_chr[ scan ] );
        scan++;
        }
    scan++;
    }

else if ( ( parse_int[ scan ] == '{' ) ||
          ( parse_int[ scan ] == '<' ) )
    {
    if ( parse_int[ scan ] == '{' )
        {
        kind = T_SET;
        c = '}';
        }
    else
        {
        kind = T_SEQUENCE;
        c = '>';
        }

    head = last = NIL_ST_REC;

    while ( parse_int[ scan ] != c )
        {
        scan++;
        rc = Build_st_value( );

        if ( last == NIL_ST_REC )
            head = rc;
        else
            last->next_ptr = rc;

        last = rc;
        }
    scan++;

    rc = Make_st_rec( NULL_STR , kind , head , NIL_ST_REC );
    }

else if ( parse_int[ scan ] == '(' )
    {
    if ( parse_buf[ scan+1 ] == ALPH )
        {
        /* disjoint union value */

        kind = ++scan;
        scan = scan + 2;
        head = Build_st_value( );
```



```
        rc = Make_st_rec( NULL_STR , T_UNION , head , NIL_ST_REC );
        (void) strcpy( head->name , parse_chr[ kind ] );
        scan++;
        }
    else
        {
        /* mask a list of all attribute or tag values */

        head = last = NIL_ST_REC;

        while ( parse_int[ scan ] != ')' )
            {
            scan++;
            rc = Build_st_value( );

            if ( last == NIL_ST_REC )
                head = rc;
            else
                last->next_ptr = rc;

            last = rc;
            }
        scan++;

        rc = Make_st_rec( NULL_STR , T_TUPLE , head , NIL_ST_REC );
        }
    }
}

else
    Error( "Build_st_value: value not defined" );

return( rc );
} /* Build_st_value */


void Add_st_var( vn , tn )
    int     vn;                                     /* variable name */
    int     tn;                                     /* type name */
    {
    struct  st_rec  *t_ptr;                         /* record pointer */

    /* check if variable and type exist */

    if (( t_ptr = Find_st_name( var_head , vn )) == NIL_ST_REC )
        if (( t_ptr = Find_st_name( type_head , tn )) == NIL_ST_REC )
            Error( "Undefined type" );
        else
            var_head = Make_st_rec( parse_chr[vn] , T_TYPE , t_ptr , var_head );

    else
        Error( "Variable already defined" );
    } /* Add_st_var */


void Add_st_type( tn , decl )
```

```
    int     tn;                                         /* type name */
    int     decl;                                /* declaration index */
    {
    struct  st_rec  *t_ptr;                          /* type pointer */

    /* check if type exists */

    if (( t_ptr = Find_st_name( type_head , tn )) == NIL_ST_REC ),
        {
        scan = decl;
        t_ptr = Build_st_type( tn );
        if ( t_ptr != NIL_ST_REC )
            {
            /* valid type - link into type list */
            t_ptr->next_ptr = type_head;
            type_head = t_ptr;
            }
        }

    else
        Error( "Type already defined" );
    } /* Add_st_type */


struct st_rec *Get_var_type( v )
    int     v;                                   /* variable index */
    {
    int     i , f;                              /* counter and flag */
    struct  st_rec  *t_ptr;                        /* list pointer */

    /* check for set and sequence values */

    if ( parse_buf[ v ] != ALPH )
        {
        scan = v;
        t_ptr = Build_st_value( );
        }

    else
        if ( ( t_ptr = Find_st_name( var_head , v ) ) == NIL_ST_REC )
            Error( "Undefined variable" );

        else
            {
            t_ptr = t_ptr->base_ptr;

            /* check each attribute and tag of qualified names */

            while ( ( parse_buf[ ++v ] == PUNCT ) &&
                    (( parse_int[ v ] == '.' ) ||
                     ( parse_int[ v ] == ';' )) )
                {
                if ( parse_int[ v ] == '.' )
                    i = T_TUPLE;
                else
                    i = T_UNION;
```



```
                f = FN_FAIL;
                v++;
                while ( ( t_ptr != NIL_ST_REC ) && ( t_ptr->kind != i ) )
                    t_ptr = t_ptr->base_ptr;
                if ( t_ptr != NIL_ST_REC )
                    t_ptr = t_ptr->base_ptr;
                while ( t_ptr != NIL_ST_REC )
                    {
                    if ( strcmp( t_ptr->name , parse_chr[ v ] ) == 0 )
                        {
                        /* found an attribute or tag */

                        f = FN_OK;
                        break;
                        }

                    t_ptr = t_ptr->next_ptr;
                    }

                if ( f == FN_FAIL )
                    {
                    if ( parse_int[ --v ] == ':' )
                        Error( "Undefined attribute" );
                    else
                        Error( "Undefined tag" );

                    return( NIL_ST_REC );
                    }
                }
            scan = v;
            }
    return( t_ptr );
    } /* Get_var_type */


void Analyze_operation( )
    {
    int     i , q;                                          /* counter */
    struct  st_rec  *trg_ptr,*src_ptr[MAX_SOURCE]; /* target and source type */
    struct  st_rec  *q_ptr[MAX_QUAL];                /* qualification types */
    int             t;                                /* type name pointer */

    for ( i = 0 ; i < MAX_QUAL ; i++ )
        q_ptr[ i ] = NIL_ST_REC;

    /* Check variable type */

    if ( ( trg_ptr = Get_var_type( target ) ) == NIL_ST_REC )
        return;

    else if ( ( trg_ptr->kind != T_SET ) && ( trg_ptr->kind != T_SEQUENCE ) ),
        {
        Error( "Target variable must be set or sequence" );
        return;
        }
```

```
/* Check each source variable */

q = 0;

for ( i = 0 ; i < MAX_SOURCE ; i++ )
    if ( source[ i ] != 0 )
        if ( ( operator == MODIFY ) ||
             (( i == 1 ) && ( operator == SELECT )) )
            {
            /* Check Qualification */

            scan = source[ i ];

            if ( operator == MODIFY )
                t = target;
            else
                t = source[ 0 ];

            if ( Analyze_qual( t, &q_ptr[ q ], &q_ptr[ q+1 ] ) == FN_FAIL )
                return;

            while ( ( parse_buf[ scan ] == RESRV ) &&
                    (( parse_int[ scan ] == AND ) ||
                     ( parse_int[ scan ] == OR )) )
                {
                scan++;
                q += 2;

                if ( Analyze_qual( t , &q_ptr[ q ] , &q_ptr[ q+1 ] )
                     == FN_FAIL )
                    return;
                }
            q += 3;
            }

        else
            if ( ( src_ptr[i] = Get_var_type( source[i] ) ) == NIL_ST_REC )
                return;
            else if ( ( src_ptr[i]->kind != T_SET ) &&
                      ( src_ptr[i]->kind != T_SEQUENCE ) )
                {
                Error( "Source variable must be set or sequence" );
                return;
                }
            else if ( operator == PROJECT )
                {
                if ( Check_project( trg_ptr , src_ptr[0] ) == FN_FAIL )
                    return;
                }
            else if ( operator == PRODUCT )
                {
                if ( ( i == 1 ) &&
                     ( Check_product( trg_ptr , src_ptr[0] , src_ptr[1] )
                       == FN_FAIL ) )
                    return;
```



```
                }
            else if ( Check_type( trg_ptr , src_ptr[i] ) == FN_FAIL )
                {
                Error( "Target and source variables must be same type" );
                return;
                }

    switch( operator )
        {
        case PRINT:
            (void) Print( trg_ptr , (char *) NULL , '\n' , 0 );
            break;

        case INSERT:
            Sl_insert( trg_ptr->base_ptr , src_ptr[ 0 ]->base_ptr , FN_END );
            break;

        case DELETE:
            Sl_delete( trg_ptr->base_ptr , src_ptr[ 0 ]->base_ptr , FN_END );
            break;

        case UNION:
            Union( trg_ptr , src_ptr );
            break;

        case DIFFERENCE:
            if ( trg_ptr == src_ptr[ 1 ] )
                Error( "Source variable is target" );
            else
                Difference( trg_ptr , src_ptr );
            break;

        case PROJECT:
            Project( trg_ptr , src_ptr[ 0 ] );
            break;

        case PRODUCT:
            Product( trg_ptr , src_ptr );
            break;

        case MODIFY:
            Modify( trg_ptr , q_ptr , source );
            break;

        case SELECT:
            if ( trg_ptr == src_ptr[ 0 ] )
                Error( "Source variable is target" );
            else
                Select( trg_ptr , q_ptr , src_ptr[ 0 ] , source );
            break;

        default:
            break;
        } /* switch */

} /* Analyze_operation */
```

```
int Analyze_qual( t , q0_ptr , ql_ptr )
    int             t;                          /* type name pointer */
    struct  st_rec  **q0_ptr;                   /* left qualifier pointer */
    struct  st_rec  **ql_ptr;                   /* right qualifier pointer */
    {
    int     b , i , k;                          /* operator/condition types */
    int     q0 , ql;                           /* qualification buffer pointers */

    if ( ( parse_buf[ scan ] == RESRV ) && ( parse_int[ scan ] == NOT ) )
        scan++;

    /* get types of both qualifiers and check type */

    if ( ( *q0_ptr = Get_var_type( q0 = scan ) ) == NIL_ST_REC )
        return( FN_FAIL );

    /* save operator/condition */

    b = parse_buf[ scan ];
    i = parse_int[ scan ];
    k = (*q0_ptr)->kind;
    scan++;

    if ( ( *ql_ptr = Get_var_type( ql = scan ) ) == NIL_ST_REC )
        return( FN_FAIL );

    if ( Check_type( *q0_ptr , *ql_ptr ) == FN_FAIL )
        {
        Error( "Qualifier arguments must be same type");
        return( FN_FAIL );
        }

    /* check that at least one argument in variable */

    if ( ( (*q0_ptr)->name[ 0 ] == '\0' ) && ( (*ql_ptr)->name[ 0 ] == '\0' ) )
        {
        Error( "One argument must be variable" );
        return( FN_FAIL );
        }

    if ( ( (*q0_ptr)->name[ 0 ] != '\0' ) &&
         ( strcmp( parse_chr[ t ] , parse_chr[ q0 ] ) != 0 ) )
        {
        Error( "Argument variable name must be same as target" );
        return( FN_FAIL );
        }

    if ( ( (*ql_ptr)->name[ 0 ] != '\0' ) &&
         ( strcmp( parse_chr[ t ] , parse_chr[ ql ] ) != 0 ) )
        {
        Error( "Argument variable name must be same as target" );
        return( FN_FAIL );
        }
```



```
    /* Check operator/condition types */

    if ( ( b == PUNCT ) &&
         (( i == '+' ) || ( i == '-' ) || ( i == '*' ) || ( i == '/' )) )
        if ( ( k != T_INTEGER ) && ( k != T_REAL ) )
            {
            Error( "Operator arguments must be integer or real");
            return( FN_FAIL );
            }

    if ( ( ( b == PUNCT ) && (( i == '<' ) || ( i == '>' )) ) ||
         ( ( b == RESRV ) && (( i == GE ) || ( i == LE )) ) )
        if ( ( k != T_INTEGER ) && ( k != T_REAL ) && ( k != T_STRING ) )
            {
            Error( "Condition arguments must be integer, real, or string" );
            return( FN_FAIL );
            }

    if ( ( b == RESRV ) && ( i == SUB ) )
        if ( ( k != T_SET ) && ( k != T_SEQUENCE ) )
            {
            Error( "Condition subset arguments must be set or sequence" );
            return( FN_FAIL );
            }

    return( FN_OK );
    } /* Analyze_qual */


int Check_type( t , s )
    struct  st_rec  *t,*s;                      /* target and source pointers */
    {
    struct  st_rec  *ss;                                /* source head pointer */
    int     rc;                                         /* return code */

    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    while ( s->kind == T_TYPE )
        s = s->base_ptr;

#if DEBUG_INTERP
    fprintf( stderr , "tk=%d  sk=%d\n",t->kind,s->kind );
#endif

    if ( ( t->kind == T_UNION ) && ( s->kind == T_UNION ) )
        {
        /* check that matching tags exist */

        t = t->base_ptr;
        s = s->base_ptr;
        rc = FN_FAIL;

        while ( t != NIL_ST_REC )
            {
            ss = s;
```

```
        while ( ss != NIL_ST_REC )
            {
            if ( strcmp( t->name , ss->name ) == 0 )
                {
                rc = FN_OK;
                break;
                }
            ss = ss->next_ptr;
            }

        if ( rc == FN_OK )
            break;
        t = t->next_ptr;
        }

    if ( rc == FN_FAIL )
        {
        Error( "Undefined tag" );
        return( rc );
        }
    }

if ( t->kind == s->kind )
    if ( t->kind == T_TUPLE )
        {
        t = t->base_ptr;
        s = s->base_ptr;

        while ( t != NIL_ST_REC )
            {
            if ( s != NIL_ST_REC )
                {
                if ( Check_type( t , s ) == FN_FAIL )
                    return( FN_FAIL );
                }
            else
                return( FN_FAIL );

            t = t->next_ptr;
            s = s->next_ptr;
            }

        if ( s != NIL_ST_REC )
            return( FN_FAIL );
        else
            return( FN_OK );
        }

    else if ( ( t->kind == T_SET ) || ( t->kind == T_SEQUENCE ) )
        {
        /* check that all set elements are of same type */

        t = t->base_ptr;
        while ( t != NIL_ST_REC )
            {
            s = s->base_ptr;
```



```
            while ( s != NIL_ST_REC )
                {
                if ( Check_type( t , s ) == FN_FAIL )
                    return( FN_FAIL );
                s = s->next_ptr;
                }
            t = t->next_ptr;
            }
        return( FN_OK );
        }

    else
        {
        if ( ( t->base_ptr != NIL_ST_REC ) && ( s->base_ptr != NIL_ST_REC ))
            return( Check_type( t->base_ptr , s->base_ptr ) );

        else if ((t->base_ptr == NIL_ST_REC) && (s->base_ptr == NIL_ST_REC))
            return( FN_OK );

        else
            return( FN_FAIL );
        }

    else
        return( FN_FAIL );

    } /* Check_type */

int Check_project( t, s )
    struct  st_rec  *t;                     /* target pointer */
    struct  st_rec  *s;                     /* source pointer */
    {
    struct  st_rec  *ss;                    /* source scan pointer */

    /* skip type declarations */

    t = t->base_ptr;
    s = s->base_ptr;

    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    while ( s->kind == T_TYPE )
        s = s->base_ptr;

    if ( ( t->kind != T_TUPLE ) || ( s->kind != T_TUPLE ) )
        {
        Error( "Target and source variables must be tuples" );
        return( FN_FAIL );
        }

    t = t->base_ptr;
    s = s->base_ptr;

    while ( t != NIL_ST_REC )
        {
```

```
    ss = s;
    while ( ss != NIL_ST_REC )
        {
        if ( strcmp( ss->name , t->name ) == 0 )
            if ( Check_type( t , ss ) == FN_FAIL )
                {
                Error("Target and source variables must be same type");
                return( FN_FAIL );
                }
            else
                break;

        ss = ss->next_ptr;
        }

    if ( ss == NIL_ST_REC )
        {
        /* attribute not found */

        Error( "Undefined attribute" );
        return( FN_FAIL );
        }
    t = t->next_ptr;
    }

    return( FN_OK );
} /* Check_project */


int Check_product( t , s0 , s1 )
    struct  st_rec  *t;                 /* target pointer */
    struct  st_rec  *s0;              /* first source pointer */
    struct  st_rec  *s1;             /* second source pointer */
    {
    struct  st_rec  *ss;              /* source scan pointer */
    struct  st_rec  *head;             /* type head pointer */

    /* skip type declarations */

    t = t->base_ptr;
    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    s0 = s0->base_ptr;
    while ( s0->kind == T_TYPE )
        s0 = s0->base_ptr;

    s1 = s1->base_ptr;
    while ( s1->kind == T_TYPE )
        s1 = s1->base_ptr;

    if ( t->kind != T_TUPLE )
        {
        Error( "Target variable must be tuple" );
        return( FN_FAIL );
        }
```



```
    t = t->base_ptr;
    ss = s0;
    head = s0;

    /* check that sum of source types same as target type */

    while ( t != NIL_ST_REC )
        {
        if ( ( ss->kind == T_TUPLE ) && ( head == ss ) )
            ss = ss->base_ptr;

        if ( Check_type( t , ss ) == FN_FAIL )
            {
            Error( "Target and source variables must be same type" );
            return( FN_FAIL );
            }

        /* get next target and source types */

        t = t->next_ptr;

        if ( head->kind == T_TUPLE )
            ss = ss->next_ptr;
        else
            ss = NIL_ST_REC;

        if ( ss == NIL_ST_REC )
            {
            if ( head == s0 )
                {
                ss = s1;
                s0 = NIL_ST_REC;
                head = s1;
                }
            else if ( t != NIL_ST_REC )
                {
                Error( "Incompatible product variable types" );
                return( FN_FAIL );
                }
            else
                return( FN_OK );
            }
        }
    Error( "Incompatible product variable types" );
    return( FN_FAIL );
} /* Check_product */


void Interpretter( )
    {
    switch( operator )
        {
        case VAR_DECL:
            Add_st_var( target , source[ 0 ] );
            break;
```

```
    case TYPE_DECL:
        Add_st_type( target , source[ 0 ] );
        break;

    default:
        Analyze_operation( );
        break;
    } /* switch */
} /* Interpretter */
```



```
#include    <stdio.h>
#include    <strings.h>
#include    "defines.h"
#include    "inter.h"
#include    "token.h"

extern  int     parse_buf[ MAX_PARSE ];             /* input parser buffer */
extern  int     parse_int[ MAX_PARSE ]; /* enumerated type and number buffer */
extern  int     operator;                             /* command operator */

extern void Error( );

char *Pack( ut , u , p )
    int             ut;                                   /* element type */
    union   sl_type *u;                       /* variant data record */
    char            *p;                       /* pack buffer pointer */
    {
    char    s[ MAX_PARSE ];                       /* packing buffer */
    int     i;                                        /* counter */

    /* pack based on simple type */

    if ( ( ut == T_INTEGER ) || ( ut == T_BOOLEAN ) )
        (void) sprintf( s , "|n%d|" , u->ival );

    else if ( ut == T_REAL )
        (void) sprintf( s , "|n%f|" , u->fval );

    else if ( ut == T_STRING )
        (void) sprintf( s , "|n%s|" , u->sval );

    else if ( ut == T_UNION )
        (void) sprintf( s , "|t%s|" , u->sval );

    /* copy result into pack buffer */

    i = 0;
    while ( ( *p++ = s[ i++ ] ) != '\0' );

    return( p-2 );

    } /* Pack */

char *Unpack( ut , u , p )
    int             ut;                                   /* element type */
    union   sl_type *u;                       /* variant data record */
    char            *p;                       /* pack buffer pointer */
    {
    int             i;                                        /* counter */

    /* unpack based on simple type */
```

```
    if ( ( ut == T_INTEGER ) || ( ut == T_BOOLEAN ) )
        (void) sscanf( p , "|n%d|" , &u->ival );

    else if ( ut == T_REAL )
        (void) sscanf( p , "|n%f|" , &u->fval );

    else if ( ut == T_STRING )
        (void) sscanf( p , "|n%[^|]" , u->sval );

    else if ( ut == T_UNION )
        (void) sscanf( p , "|t%[^|]" , u->sval );

    /* advance buffer pointer */

    while ( ( *p != '\0' ) && ( *p++ != '|' ) );
    while ( ( *p != '\0' ) && ( *p != '|' ) )
        p++;

    if ( ( ut == T_STRING ) || ( ut == T_UNION ) )
        {
        i = 0;
        while ( ( u->sval[ i ] != '|' ) && ( u->sval[ i ] != '\0' ) )
            i++;
        u->sval[ i ] = '\0';
        }

    return( p );

    } /* Unpack */


int Get_next_rec( t , key , data )
    struct  st_rec   *t;                                    /* type record */
    struct  data_rec *key;                                  /* key pointer */
    struct  data_rec *data;                                 /* data pointer */
    {
    char            *fn;                                    /* file name */

    /*
        Assumes Find flags as follows:

        Flag                Given key?      Return key      Return data
        ----                ----------      ----------      -----------

        MASK_FIND_LEFT      no              leftmost key    leftmost data
        MASK_FIND_NEXT      yes             next key        next data
        MASK_FIND_KEY       yes             given key       corresponding data
    */

    /* fill interface key record */

    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    /* determine file name */
    /* $$$ unshared case only */
```



```
    fn = t->name;

    /* find the requested record */

    if ( Find( key , data , fn ) <= 0 )
        return( FN_END );

    else if ( data->length == 0 )
        return( FN_END );

    else
        {
        /* ensure string is null terminated */

        key->data[ key->length ] = data->data[ data->length ] = '\0';
        return( FN_MORE );
        }
    } /* Get_next_rec */


char *Cvt_packed( t , s , p , fl )
    struct  st_rec  *t;                                     /* target pointer */
    struct  st_rec  *s;                                     /* source pointer */
    char            *p;                                     /* pack buffer pointer */
    int             fl;                                     /* nested coding flags */
    {
    struct  data_rec key_buf;                               /* key record */
    struct  data_rec data_buf;                              /* data record */

    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    while ( s->kind == T_TYPE )
        s = s->base_ptr;

    /* pack depending on kind of value */

    switch( s->kind )
        {
        case T_INTEGER:
        case T_REAL:
        case T_BOOLEAN:
        case T_STRING:
            p = Pack( s->kind , &s->val , p );
            break;

        case T_UNION:
            /* pack the tag and the value */

            t = t->base_ptr;
            s = s->base_ptr;
            p = Pack( T_UNION , (union sl_type *) s->name , p );
            p = Cvt_packed( t , s , p , fl );
            break;
```

```
    case T_SET:
    case T_SEQUENCE:
        /* pack all elements */

        t = t->base_ptr;
        s = s->base_ptr;
        while ( s != NULL )
            {
            p = Cvt_packed( t , s , p , (fl | MASK_NESTED_SET) );
            s = s->next_ptr;
            }
        *p++ = '|';
        *p = '|';
        *( p+1 ) = '\0';
        break;

    case T_TUPLE:
        if ( ( fl & MASK_NESTED_TUPLE ) == 0 )
            {
            /* pack all elements */

            t = t->base_ptr;
            s = s->base_ptr;

            while ( s != NULL )
                {
                p = Cvt_packed( t , s , p , (fl | MASK_NESTED_TUPLE) );
                t = t->next_ptr;
                s = s->next_ptr;
                }
            }
        else
            {
            /* pack then code nested tuple */

            if ( Get_code( t , s , &key_buf , &data_buf ) == FN_FAIL )
                {
                /* insert the tuple only for insert operations */

                if ( ( operator == INSERT ) || ( operator == UNION ) )
                    {
                    key_buf.flags = 0;
                    if ( Insert( &key_buf , &data_buf , t->name ) <= 0 )
                        {
                        Error( "Cvt_packed: Insert call failed" );
                        key_buf.data[ 0 ] = '\0';
                        }
                    }
                else
                    key_buf.data[ 0 ] = '\0';
                }

            (void) sscanf( key_buf.data , "%d" , &s->val.ival );
            p = Pack( T_INTEGER , &s->val , p );
            } /* if */
        break;
```



```
        } /* switch */

    return( p );

    } /* Cvt_packed */


char *Print( t , p , ln , fl )
    struct  st_rec  *t;                         /* type record */
    char            *p;                         /* record pointer */
    char            ln;                         /* line control */
    int             fl;                         /* nested coding flags */
    {
    struct  data_rec key_buf;                   /* key record */
    struct  data_rec data_buf;                  /* data record */
    char            *d_ptr;                     /* data record pointer */
    union   sl_type u;                          /* variant data type */
    char            nln;                        /* next new line */

    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    /* print based on SL type */

    switch( t->kind )
        {
        case T_INTEGER:
            p = Unpack( T_INTEGER , &u , p );
            fprintf( stderr , "%c %d " , ln , u.ival );
            break;

        case T_REAL:
            p = Unpack( T_REAL , &u , p );
            fprintf( stderr , "%c %f " , ln , u.fval );
            break;

        case T_BOOLEAN:
            p = Unpack( T_BOOLEAN , &u , p );
            if ( u.ival == 0 )
                fprintf( stderr , " %c false " , ln );
            else
                fprintf( stderr , "%c true " , ln );
            break;

        case T_STRING:
            p = Unpack( T_STRING , &u , p );
            fprintf( stderr , "%c \"%s\" " , ln , u.sval );
            break;

        case T_UNION:
            /* unpack the tag and then the corresponding value */

            p = Unpack( T_UNION , &u , p );
            fprintf( stderr , "%c { %s " , ln , u.sval );
```

```
        t = t->base_ptr;
        while ( t != NULL )
            {
            if ( strcmp( t->name , u.sval ) == 0 )
                /* found the tag */

                break;

            t = t->next_ptr;
            }

        p = Print( t , p , ',' , fl );
        fprintf( stderr , ")" , ln );
        break;

    case T_SET:
    case T_SEQUENCE:
        if ( t->kind == T_SET )
            fprintf( stderr , "%c {" , ln );
        else
            fprintf( stderr , "%c <" , ln );

        /* print uncoded sets and sequences directly */

        nln = '\0';

        if ( ( fl & MASK_NESTED_SET ) == 0 )
            {
            key_buf.flags = MASK_FIND_LEFT;
            while ( Get_next_rec( t->base_ptr , &key_buf , &data_buf )
                                == FN_MORE )
                {
                p = Print( t->base_ptr , data_buf.data , nln ,
                        (fl | MASK_NESTED_SET) );
                nln = ',';
                key_buf.flags = MASK_FIND_NEXT;
                }
            }

        else
            {
            /* just print the set or sequence */

            if ( *( p+1 ) == '|' )
                p++;

            else
                while ( *( p = Print( t->base_ptr, p, nln, fl ) ) != '\0' )
                    {
                    if ( *( p+1 ) == '|' )
                        {
                        p++;
                        break;
                        }

                    nln = ',';
```



```
                    } /* while */
                }

            if ( t->kind == T_SET )
                fprintf( stderr , "} " );
            else
                fprintf( stderr , "> " );
            break;

        case T_TUPLE:
            fprintf( stderr , "%c (" , ln );
            nln = '\0';

            /* print uncoded tuples directly */

            if ( ( fl & MASK_NESTED_TUPLE ) == 0 )
                {
                t = t->base_ptr;
                while ( *( p = Print( t , p , nln , (fl | MASK_NESTED_TUPLE) ) )
                                != '\0' )
                    {
                    nln = ',';
                    t = t->next_ptr;
                    }
                fprintf( stderr , ")\n " );
                }

            else
                {
                /* coded tuples must be uncoded first */

                p = Unpack( T_INTEGER , &u , p );
                (void) sprintf( key_buf.data , "%d" , u.ival );
                key_buf.length = 0;
                while ( key_buf.data[ key_buf.length ] != '\0' )
                    key_buf.length++;
                key_buf.flags = MASK_FIND_KEY;

                if ( Get_next_rec( t , &key_buf , &data_buf ) == FN_FAIL )
                    {
                    Error( "Print: coded tuple does not exist" );
                    return( p );
                    }

                d_ptr = data_buf.data;
                t = t->base_ptr;
                while ( *( d_ptr = Print( t , d_ptr , nln , fl ) ) != '\0' )
                    {
                    nln = ',';
                    t = t->next_ptr;
                    }
                fprintf( stderr , ") " );
                } /* if */

            break;
        } /* switch */
```

```
    if ( *( p + 1 ) == '\0' )
        p++;

    return( p );

    }/* Print */


int Get_code( t , s , key_buf , data_buf )
    struct  st_rec   *t;                                /* target pointer */
    struct  st_rec   *s;                                /* source pointer */
    struct  data_rec *key_buf;                      /* key buffer pointer */
    struct  data_rec *data_buf;                /* packed data buffer pointer */
    {
    FILE             *fp;                                /* file pointer */
    char             src_buf[ MAX_PARSE ];         /* packed source data */
    int              i;                                    /* counter */

    /* prepare source buffer */

    if ( s->kind == T_TYPE )
        {
        /* copy the record to insert to a safe place */

        i = 0;
        while ( ( src_buf[ i ] = data_buf->data[ i ] ) != '\0' )
            i++;
        }

    else
        (void) Cvt_packed( t , s , src_buf , 0 );

    /* create the relation file is it doesn't exist */

    if ( ( fp = fopen( t->name , "r" ) ) == NULL )
        close( Creat( t->name ) );
    else
        fclose( fp );

    /* check if the tuple already exists */

    key_buf->flags = MASK_FIND_LEFT;
    while ( Get_next_rec( t , key_buf , data_buf ) == FN_MORE )
        {
        if ( strcmp( data_buf->data , src_buf ) == 0 )
            return( FN_OK );

        key_buf->flags = MASK_FIND_NEXT;
        }

    /* increment the key */

    if ( key_buf->flags == MASK_FIND_LEFT )
        {
        key_buf->data[ 0 ] = '0';
```



```
        key_buf->data[ 1 ] = '\0';
        key_buf->length = 1;
        }

    else
        {
        (void) sscanf( key_buf->data , "%d" , &i );
        i++;
        (void) sprintf( key_buf->data , "%d" , i );

        key_buf->length = 0;
        while ( i >= 1 )
            {
            key_buf->length++;
            i /= 10;
            }
        }

    /* fill in packed data length */

    i = 0;
    while ( ( data_buf->data[ i ] = src_buf[ i ] ) != '\0' )
        i++;
    data_buf->length = i;

    return( FN_FAIL );

    } /* Get_code */


void Sl_insert( t , s , fl )
    struct  st_rec  *t;                                 /* insert target */
    struct  st_rec  *s;                                 /* insert source */
    int             fl;                        /* continue on error flag */
    {
    struct  data_rec key_buf;                           /* key buffer */
    struct  data_rec data_buf;                          /* data buffer */
    struct  data_rec k_buf;                             /* key buffer */
    int             more;                             /* more data flag */

    /* check whether source is variable or value */

    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    if ( s->kind == T_TYPE )
        {
        /* read first record of variable */

        key_buf.flags = MASK_FIND_LEFT;
        more = Get_next_rec( s , &key_buf , &data_buf );
        }

    else
        more = FN_MORE;
```

```
    /* insert all records given */

    while ( more == FN_MORE )
        {
        /* check if the record exists and obtain a key */

        if ( Get_code( t , s , &k_buf , &data_buf ) == FN_OK )
            {
            if ( fl == FN_END )
                {
                Error( "Tuple value already exists" );
                return;
                }
            }
        else
            {
            /* insert the record */

            k_buf.flags = 0;
            if ( Insert( &k_buf , &data_buf , t->name ) <= 0 )
                {
                Error( "Sl_insert: Insert call failed" );
                return;
                }
            }

        /* get next record */

        if ( s->kind == T_TYPE )
            {
            key_buf.flags = MASK_FIND_NEXT;
            more = Get_next_rec( s , &key_buf , &data_buf );
            }
        else
            {
            s = s->next_ptr;
            if ( s == NULL )
                more = FN_END;
            }

        } /* while */
    }/* Sl_insert */


void Sl_delete( t , s , fl )
    struct st_rec  *t;                          /* delete target */
    struct st_rec  *s;                          /* delete source */
    int            fl;                        /* break on error flag */
    {
    struct data_rec key_buf;                    /* key buffer */
    struct data_rec data_buf;                   /* data buffer */
    struct data_rec k_buf;                      /* key buffer */
    int             more;                     /* more data flag */
    int             rc;                         /* return code */

    /* check whether source is variable or value */
```



```
    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    if ( s->kind == T_TYPE )
        {
        /* read first record of variable */

        key_buf.flags = MASK_FIND_LEFT;
        more = Get_next_rec( s , &key_buf , &data_buf );
        }

    else
        more = FN_MORE;

    /* delete all records given */

    while ( more == FN_MORE )
        {
        /* check if the record exists and obtain a key */

        if ( ( ( rc = Get_code( t , s , &k_buf , &data_buf ) ) == FN_FAIL ) &&
             ( fl == FN_END) )
            {
            Error( "Tuple value does not exist" );
            return;
            }

        /* get next record */

        if ( s->kind == T_TYPE )
            {
            key_buf.flags = MASK_FIND_NEXT;
            more = Get_next_rec( s , &key_buf , &data_buf );
            }
        else
            {
            s = s->next_ptr;
            if ( s == NULL )
                more = FN_END;
            }

        /* delete the new record */

        if ( rc == FN_OK )
            if ( Delete( &k_buf , &data_buf , t->name ) <= 0 )
                {
                Error( "Sl_delete: Delete call failed" );
                return;
                }

        } /* while */
    }/* Sl_delete */

void Union( t , s )
```

```
    struct  st_rec  *t;                             /* target pointer */
    struct  st_rec  *s[ MAX_SOURCE ];               /* source pointers */

    int             done[ MAX_SOURCE ];         /* source complete flags */
    int             i;                                      /* counter */
    char            sys[ MAX_PARSE ];               /* system call buffer */

    /* don't consider a source file which is also a target */

    t = t->base_ptr;

    for ( i = 0 ; i < MAX_SOURCE ; i++ )
        {
        s[ i ] = s[ i ]->base_ptr;
        done[ i ] = ( s [ i ]->kind == T_TYPE ) &&
                    (strcmp( s[ i ]->base_ptr->name , t->base_ptr->name ) == 0);
        }

    /* otherwise if a variable participates then just copy it */

    if ( ( done[ 0 ] == 0 ) && ( done[ 1 ] == 0 ) )
        for ( i = 0 ; i < MAX_SOURCE ; i++ )
            if ( s [ i ]->kind == T_TYPE )
                {
                done[ i ] = 1;
                (void) sprintf( sys , "cp %s %s" ,
                                s[ i ]->base_ptr->name , t->base_ptr->name );
                system( sys );
                break;
                }

    /* if two values participate then reset the file */

    if ( ( done[ 0 ] == 0 ) && ( done[ 1 ] == 0 ) )
        close( Creat( t->base_ptr->name ) );

    /* insert remaining records */

    for ( i = 0 ; i < MAX_SOURCE ; i++ )
        if ( done[ i ] == 0 )
            Sl_insert( t , s[ i ] , FN_MORE );

    } /* Union */


void Difference( t , s )
    struct st_rec  *t;                              /* target pointer */
    struct st_rec  *s[ MAX_SOURCE ];                /* source pointers */
    {
    char            sys[ MAX_PARSE ];               /* system call buffer

    /* advance base pointers */

    t = t->base_ptr;
    s[ 0 ] = s[ 0 ]->base_ptr;
    s[ 1 ] = s[ 1 ]->base_ptr;
```



```
    /* move first source to target */

    if ( s[ 0 ]->kind == T_TYPE )
        {
        /* degenerates to a delete when target and first source are same */

        if ( strcmp( s[ 0 ]->base_ptr->name , t->base_ptr->name ) != 0 )
            {
            (void) sprintf( sys , "cp %s %s" ,
                            s[ 0 ]->base_ptr->name , t->base_ptr->name );
            system( sys );
            }
        }
    else
        {
        /* reset target with values */

        close( Creat( t->base_ptr->name ) );
        Sl_insert( t , s[ 0 ] , FN_END );
        }

    /* Now delete the remaining records */

    Sl_delete( t , s [ 1 ] , FN_MORE );

    } /* Difference */


char *Attrib( s , d )
    struct st_rec  *s;                              /* source pointer */
    char           *d;                              /* buffer pointer */
    {
    /* advance source pointer */

    while ( s->kind == T_TYPE )
        s = s->base_ptr;

    /* unpack based on kind of value */

    switch ( s->kind )
        {
        case T_INTEGER:
        case T_REAL:
        case T_BOOLEAN:
        case T_STRING:
            d = Unpack( s->kind , &s->val , d );
            break;

        case T_UNION:
            /* unpack tag and value */

            s = s->base_ptr;
            d = Unpack( T_UNION , (union sl_type *) s->name , d );
            d = Attrib( s , d );
            break;
```

```
        case T_SET:
        case T_SEQUENCE:
            /* unpack each value */

            if ( *( d + 1 ) == '|' )
                d++;

            else
                {
                s = s->base_ptr;
                while ( *( d = Attrib( s , d ) ) != '\0' )
                    if ( *( d + 1 ) == '|' )
                        {
                        d++;
                        break;
                        }
                ) /* if */

            break;

        case T_TUPLE:
            /* unpack the code */

            d = Unpack( T_INTEGER , &s->val , d );
            break;

        } /* switch */

        return( d );

    } /* Attrib */


char *Get_attrib( s , a , data , p )
    struct  st_rec  *s;                         /* source tuple pointer */
    struct  st_rec  *a;                 /* source tuple attribute pointer */
    char            *data;                       /* data buffer pointer */
    char            *p;                          /* pack buffer pointer */
    {
    char            *d;                         /* pointer for scanning */

    /* advance source pointer to tuple base type */

    while ( s->kind == T_TYPE )
        s = s->base_ptr;

    s = s->base_ptr;

    /* unpack data buffer looking for the attribute */

    while ( s != NULL )
        {
        d = Attrib( s , data );

        if ( strcmp( s->name , a->name ) == 0 )
```



```
            /* found attribute */

            break;

        data = d;
        s = s->next_ptr;
        ) /* while */

    /* return the data for the attribute */

    while ( data != d )
        *p++ = *data++;

    *p = *data;
    *( p + 1 ) = '\0';

    return( p );

    ) /* Get_attrib */


void *Set_attrib( s , a , data )
    struct  st_rec  *s;                         /* source tuple pointer */
    struct  st_rec  *a;                 /* source tuple attribute pointer */
    char            *data;                       /* data buffer pointer */
    {
    char            *d, *hd;                    /* pointers for scanning */
    char            buf[ MAX_PARSE ];                    /* new attribute */
    char            d_buf[ MAX_PARSE ];                  /* new buffer */

    /* advance source pointer to tuple base type */

    while ( s->kind == T_TYPE )
        s = s->base_ptr;

    s = s->base_ptr;

    /* save the new attribute in a safe place */

    (void) strcpy( buf , a->val.sval );
    hd = data;

    /* unpack data buffer looking for the attribute */

    while ( s != NULL )
        {
        d = Attrib( s , data );

        if ( strcmp( s->name , a->name ) == 0 )
            /* found attribute */

            break;

        data = d;
        s = s->next_ptr;
        ) /* while */
```

```
    /* replace the attribute with the new value */

    *data = '\0';
    (void) strcpy( d_buf , hd );
    (void) strcat( d_buf , buf );
    (void) strcat( d_buf , ++d );
    (void) strcpy( hd , d_buf );

    } /* Set_attrib */


void Project( t , s )
    struct  st_rec  *t;                                 /* target pointer */
    struct  st_rec  *s;                                 /* source pointer */
    {
    struct  data_rec key_buf , data_buf;        /* source record key and data */
    struct  data_rec k_buf , d_buf;             /* target record key and data */
    struct  st_rec   *tt , *ss;           /* target and source scan pointers */
    char            *p;                             /* data buffer pointer */

    /* advance pointers and clear target relation */

    if ( t == s )
        return;

    t = t->base_ptr;
    s = s->base_ptr;

    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    close( Creat( t->name ) );

    /* retrieve all records in source */

    key_buf.flags = MASK_FIND_LEFT;
    while ( Get_next_rec( s , &key_buf , &data_buf ) == FN_MORE )
        {
        p = d_buf.data;
        tt = t->base_ptr;

        /* select only the projected attributes */

        while ( tt != NULL )
            {
            ss = s;
            while ( ss->kind == T_TYPE )
                ss = ss->base_ptr;
            ss = ss->base_ptr;

            while ( ss != NULL )
                {
                if ( strcmp( ss->name , tt->name ) == 0 )
                    {
                    /* found a projected attribute */
```



```
                    p = Get_attrib( s , ss , data_buf.data , p );
                    break;
                    } /* if */

                ss = ss->next_ptr;
                } /* while */

            tt = tt->next_ptr;
            } /* while */

        /* now add the projected tuple to the target */

        if ( Get_code( t , s , &k_buf , &d_buf ) == FN_FAIL )
            {
            k_buf.flags = 0;
            if ( Insert( &k_buf , &d_buf , t->name ) <= 0 )
                {
                Error( "Project: Insert call failed" );
                return;
                } /* if */
            }

        key_buf.flags = MASK_FIND_NEXT;
        }
    } /* Project */


void Product( t , s )
    struct  st_rec  *t;                                 /* target pointer */
    struct  st_rec  *s[ MAX_SOURCE ];                   /* source pointers */
    {
    struct  data_rec k_buf , key_buf;                   /* key buffers */
    struct  data_rec d0_buf , data_buf;                 /* data buffers */
    struct  data_rec d1_buf[ MAX_RECORDS ];     /* inner operand data */
    struct  st_rec   *ss;                       /* inner operand pointer */
    int             more;                       /* merging flags */
    int             i , ii;                     /* inner operand count */

    /* advance base pointers */

    t = t->base_ptr;
    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    s[ 0 ] = s[ 0 ]->base_ptr;
    s[ 1 ] = s[ 1 ]->base_ptr;

    /* reset the target file */

    close( Creat( t->name ) );

    /* prepare inner operand to speed operation */

    ii = 0;
    ss = s[ 1 ];
```

```
    if ( ss->kind == T_TYPE )
        {
        k_buf.flags = MASK_FIND_LEFT;
        more = Get_next_rec( ss , &k_buf , &dl_buf[ ii ] );
        }
    else
        {
        (void) Cvt_packed( t , ss , dl_buf[ ii ].data , 0 );
        more = FN_MORE;
        }

    while ( more == FN_MORE )
        {
        ii++;

        if ( ss->kind == T_TYPE )
            {
            k_buf.flags = MASK_FIND_NEXT;
            more = Get_next_rec( ss , &k_buf , &dl_buf[ ii ] );
            }
        else
            {
            ss = ss->next_ptr;
            if ( ss == NULL )
                more = FN_END;
            else
                (void) Cvt_packed( t , ss , dl_buf[ ii ].data , 0 );
            }
        } /* while */

    /* use all records in outer operand */

    if ( s[ 0 ]->kind == T_TYPE )
        {
        k_buf.flags = MASK_FIND_LEFT;
        more = Get_next_rec( s[ 0 ] , &k_buf , &d0_buf );
        }
    else
        {
        (void) Cvt_packed( t , s[ 0 ] , d0_buf.data , 0 );
        more = FN_MORE;
        }

    s[ 1 ]->kind = T_TYPE;
    while ( more == FN_MORE )
        {
        /* use all records in inner operand */

        i = 0;
        while ( i < ii )
            {
            /* combine records and insert in target */

            data_buf.data[ 0 ] = '\0';
            (void) strcat( data_buf.data , d0_buf.data );
```



```
            (void) strcat( data_buf.data , &dl_buf[ i ].data[ 1 ] );

            if ( Get_code( t , s[ 1 ] , &key_buf , &data_buf ) == FN_FAIL )
                {
                key_buf.flags = 0;
                if ( Insert( &key_buf , &data_buf , t->name ) <= 0 )
                    {
                    Error( "Product: Insert call failed" );
                    return;
                    }
                }
            i++;
            } /* while */

        if ( s[ 0 ]->kind == T_TYPE )
            {
            k_buf.flags = MASK_FIND_NEXT;
            more = Get_next_rec( s[ 0 ] , &k_buf , &d0_buf );
            }
        else
            {
            s[ 0 ] = s[ 0 ]->next_ptr;
            if ( s[ 0 ] == NULL )
                more = FN_END;
            else
                (void) Cvt_packed( t , s[ 0 ] , d0_buf.data , 0 );
            }
        } /* while */

    } /* Product */


void Operator( op , t , src , dst , d )
    char            op;                                 /* operator */
    struct  st_rec  *t;                                 /* target pointer */
    struct  st_rec  *src;                               /* source pointer */
    struct  st_rec  *dst;                               /* destination pointer */
    char            *d;                                 /* buffer pointer */
    {
    char            buf[ MAX_PARSE ];                   /* pack buffer */
    char            *b , *bb;                           /* buffer pointers */

    /* find the variable attributes and convert to a value */

    if ( src->name[ 0 ] != '\0' )
        {
        b = Get_attrib( t , src , d , buf );
        b = Unpack( src->kind , &src->val , buf );
        }

    if ( dst->name[ 0 ] != '\0' )
        {
        b = Get_attrib( t , dst , d , buf );
        b = Unpack( dst->kind , &dst->val , buf );
        }
```

```
    /* now do the operation */

    if ( dst->kind == T_INTEGER )
        {
        /* integer operations */

        if ( op == '+' )
            dst->val.ival += src->val.ival;
        else if ( op == '-' )
            dst->val.ival -= src->val.ival;
        else if ( op == '*' )
            dst->val.ival *= src->val.ival;
        else if ( op == '/' )
            dst->val.ival /= src->val.ival;
        }

    else if ( dst->kind == T_REAL )
        {
        /* real operations */

        if ( op == '+' )
            dst->val.fval += src->val.fval;
        else if ( op == '-' )
            dst->val.fval -= src->val.fval;
        else if ( op == '*' )
            dst->val.fval *= src->val.fval;
        else if ( op == '/' )
            dst->val.fval /= src->val.fval;
        }

    /* pack the result and save in the destination */

    b = Pack( dst->kind , &dst->val , buf );

    b = buf;
    bb = dst->val.sval;
    while ( ( *bb++ = *b++ ) != '\0' );

    } /* Operator */


void Modify( t , q_ptr , s )
    struct  st_rec  *t;                             /* target pointer */
    struct  st_rec  *q_ptr[ MAX_QUAL ];             /* qualification pointers */
    int             s[ MAX_SOURCE ];        /* command source variable index */
    {
    struct  data_rec key_buf;                               /* key record */
    struct  data_rec data_buf;                              /* data record */
    struct  data_rec old_buf;       /* $$$ old data buffer for delete/insert */
    int             i , q , qq;                             /* counters */
    int             more;                                   /* flag */

    t = t->base_ptr;
    while ( t->kind == T_TYPE )
        t = t->base_ptr;
```



```
    qq = 0;
    while ( q_ptr[ qq++ ] != NULL );

    /* modify all records */

    key_buf.flags = MASK_FIND_LEFT;
    while ( Get_next_rec( t , &key_buf , &data_buf ) == FN_MORE )
        {
        /* $$$ save the old buffer for delete/insert */

        old_buf.length = data_buf.length;
        (void) strncpy( old_buf.data , data_buf.data , data_buf.length );

        /* use all modification clauses */

        i = s[ 0 ] + 1;
        q = 0;
        more = FN_MORE;
        while ( ( more == FN_MORE ) &&
                ( ( s[ 1 ] == 0 ) ||
                ( Selected( t , q_ptr , s[1] , qq , data_buf.data ) == FN_OK )))
            {
            /* skip to operator */

            while ( ( parse_buf[ i ] != PUNCT ) ||
                    (( parse_int[ i ] != '+' ) && ( parse_int[ i ] != '-' ) &&
                     ( parse_int[ i ] != '*' ) && ( parse_int[ i ] != '/' )) )
                i++;

            /* determine the source and destination and replace attribute */

            if ( q_ptr[ q ]->name[ 0 ] == '\0' )
                {
                Operator( parse_int[ i ] , t , q_ptr[ q ] , q_ptr[ q+1 ] ,
                          data_buf.data );
                Set_attrib( t , q_ptr[ q+1 ] , data_buf.data );
                }
            else
                {
                Operator( parse_int[ i ] , t , q_ptr[ q+1 ] , q_ptr[ q ] ,
                          data_buf.data );
                Set_attrib( t , q_ptr[ q ] , data_buf.data );
                }

            /* check for more modifications */

            q += 2;

            if ( q_ptr[ q ] != NULL )
                {
                while ( parse_buf[ ++i ] != RESRV );
                if ( ( parse_int[ i ] == AND ) || ( parse_int[ i ] == OR ) )
                    i += 2;
                else
                    more = FN_END;
                }
```

```
        else
            more = FN_END;
        } /* while */

    /* replace the modified tuple */

    if ( more == FN_END )
        {
        /* %%% the following code is used when INS_REPLACE is supported */
        /*
        key_buf.flags = MASK_INS_REPLACE;
        data_buf.length = 0;
        while ( data_buf.data[ data_buf.length ] != '\0' )
            data_buf.length++;

        if ( Insert( &key_buf , &data_buf , t->name ) <= 0 )
            {
            Error( "Modify: Insert call failed" );
            return;
            }
        */
        /* %%% for insert/delete, the following is used */

        if ( Delete( &key_buf , &old_buf , t->name ) <= 0 )
            {
            Error( "Modify: Delete call failed" );
            return;
            }

        key_buf.flags = 0;
        data_buf.length = 0;
        while ( data_buf.data[ data_buf.length ] != '\0' )
            data_buf.length++;

        if ( Insert( &key_buf , &data_buf , t->name ) <= 0 )
            {
            Error( "Modify: Insert call failed" );
            return;
            }
        } /* if */

    key_buf.flags = MASK_FIND_NEXT;
    } /* while */

} /* Modify */


int Condition( i , t , q0 , ql , d )
    int             i;                          /* parse buffer index */
    struct  st_rec  *t;                         /* target pointer */
    struct  st_rec  *q0;                        /* left qualification */
    struct  st_rec  *ql;                        /* right qualification */
    char            *d;                         /* data buffer pointer */
    {
    char            buf0[ MAX_PARSE ] , buf1[ MAX_PARSE ];  /* pack buffers */
    struct  st_rec  *tt;                        /* attribute pointer */
```



```
    int             rc;                         /* return code */

    /* get the packed representation of the arguments */

    if ( q0->name[ 0 ] == '\0' )
        {
        tt = t->base_ptr;
        while ( strcmp( tt->name , q1->name ) != 0 )
            tt = tt->next_ptr;

        (void) Cvt_packed( tt , q0 , buf0 , MASK_NESTED_TUPLE );
        }
    else
        (void) Get_attrib( t , q0 , d , buf0 );

    if ( q1->name[ 0 ] == '\0' )
        {
        tt = t->base_ptr;
        while ( strcmp( tt->name , q0->name ) != 0 )
            tt = tt->next_ptr;

        (void) Cvt_packed( tt , q1 , buf1 , MASK_NESTED_TUPLE );
        }
    else
        (void) Get_attrib( t , q1 , d , buf1 );

    /* evaluate the condition */

    if ( ( parse_buf[ i ] == RESRV ) && ( parse_int[ i ] == SUB ) )
        /* subset condition */

        return( 0 );

    else
        {
        if ( ( q0->kind == T_INTEGER ) || ( q0->kind == T_REAL ) )
            {
            if ( q0->name[ 0 ] != '\0' )
                (void) Unpack( q0->kind , &q0->val , buf0 );

            if ( q1->name[ 0 ] != '\0' )
                (void) Unpack( q1->kind , &q1->val , buf1 );
            }

        else
            rc = strcmp( buf0 , buf1 );

        if ( parse_buf[ i ] == PUNCT )
            {
            if ( parse_int[ i ] == '=' )
                return( rc == 0 );
            else if ( parse_int[ i ] == '<' )
                {
                if ( q0->kind == T_INTEGER )
                    return( q0->val.ival < q1->val.ival );
                else if ( q0->kind == T_REAL )
```

```
                return( q0->val.fval < ql->val.fval );
            else
                return( rc < 0 );
            }
        else
            {
            if ( q0->kind == T_INTEGER )
                return( q0->val.ival > ql->val.ival );
            else if ( q0->kind == T_REAL )
                return( q0->val.fval > ql->val.fval );
            else
                return( rc > 0 );
            }
        }

    else
        {
        if ( parse_int[ i ] == NE )
            return( rc != 0 );
        else if ( parse_int[ i ] == LE )
            {
            if ( q0->kind == T_INTEGER )
                return( q0->val.ival <= ql->val.ival );
            else if ( q0->kind == T_REAL )
                return( q0->val.fval <= ql->val.fval );
            else
                return( rc <= 0 );
            }
        else
            {
            if ( q0->kind == T_INTEGER )
                return( q0->val.ival >= ql->val.ival );
            else if ( q0->kind == T_REAL )
                return( q0->val.fval >= ql->val.fval );
            else
                return( rc >= 0 );
            }
        }
    } /* if */

} /* Condition */


int Selected( t , q_ptr , s , q , d )
    struct  st_rec  *t;                     /* target pointer */
    struct  st_rec  *q_ptr[ MAX_QUAL ];     /* qualification pointer */
    int             s;                      /* source buffer index */
    int             q;                      /* source qualification index */
    char            *d;                     /* data buffer pointer */
    {
    int             i;                      /* index */
    int             rc , r;                 /* return codes */
    int             more , con;             /* flags */

    /* evaluate each qualification */
```



```
    rc = FN_MORE;
    con = AND;
    i = s;
    more = FN_MORE;

    while ( more == FN_MORE )
        {
        if ( ( parse_buf[ i ] == RESRV ) && ( parse_int[ i ] == NOT ) )
            i++;

        /* find the condition */

        while ( ( parse_buf[ i ] != PUNCT ) ||
                ( ( parse_int[ i ] != '=' ) && ( parse_int[ i ] != '<' ) &&
                  ( parse_int[ i ] != '>' ) ) )
            if ( ( parse_buf[ i ] == SPEC ) &&
                 ( ( parse_int[ i ] == NE ) || ( parse_int[ i ] == LE ) ||
                   ( parse_int[ i ] == GE ) ) )
                break;
            else if ( ( parse_buf[ i ] == RESRV ) && ( parse_int[ i ] == SUB ) )
                break;
            else
                i++;

        r = Condition( i , t , q_ptr[ q ] , q_ptr[ q+1 ] , d );

        /* check whether result should be negated */

        if ( ( parse_buf[ s ] == RESRV ) && ( parse_int[ s ] == NOT ) )
            r = r == 0;

        if ( rc == FN_MORE )
            rc = r;

        if ( con == AND )
            rc = rc && r;
        else
            rc = rc || r;

        q += 2;

        /* look for a connective if one exists */

        if ( q_ptr[ q ] != NULL )
            {
            while ( ( parse_buf[ i ] != RESRV ) ||
                    ( ( parse_int[ i ] != AND ) && ( parse_int[ i ] != OR ) ) )
                i++;

            con = parse_int[ i++ ];
            }
        else
            more = FN_END;

        s = i;
        } /* while */
```

```
    if ( rc == 0 )
        rc = FN_FAIL;
    else
        rc = FN_OK;

    return( rc );

    } /* Selected */


void Select( t , q_ptr , s , ss )
    struct  st_rec  *t;                         /* target pointer */
    struct  st_rec  *q_ptr[ MAX_QUAL ];         /* qualification pointers */
    struct  st_rec  *s;                         /* source pointer */
    int             ss[ MAX_SOURCE ];           /* parse buffer index */
    {
    struct  data_rec  key_buf , k_buf;          /* key records */
    struct  data_rec  data_buf;                 /* data records */
    struct  st_rec    *st;                      /* source type pointer */

    /* advance target and source pointers */

    t = t->base_ptr;
    while ( t->kind == T_TYPE )
        t = t->base_ptr;

    st = s = s->base_ptr;
    while ( s->kind == T_TYPE )
        s = s->base_ptr;

    close( Creat( t->name ) );
    key_buf.flags = MASK_FIND_LEFT;

    /* examine all records to check whether they will be selected */

    while ( Get_next_rec( s , &key_buf , &data_buf ) == FN_MORE )
        {
        if ( Selected( s , q_ptr , ss[ 1 ] , 0 , data_buf.data ) == FN_OK )
            if ( Get_code( t , st , &k_buf , &data_buf ) == FN_FAIL )
                {
                k_buf.flags = 0;
                if ( Insert( &k_buf , &data_buf , t->name ) <= 0 )
                    {
                    Error( "Select: Insert call failed" );
                    return;
                    }
                }
        key_buf.flags = MASK_FIND_NEXT;
        }

    }/* Select */
```



```
#include    <strings.h>
#include    <stdio.h>
#include    "defines.h"
#include    "inter.h"

int Pind( key , buf , fn )
    struct  data_rec    *key;                   /* keyword key */
    struct  data_rec    *buf;                   /* returned data */
    char                *fn;                    /* file name */
    {

    FILE    *fp;                                /* stream file pointer */
    char    b[ MAX_INTER_DATA ];                /* temporary input buffer */
    int     i , rc;                             /* return code */

    if ( ( fp = fopen( fn , "r" ) ) == NULL )
        return( 0 );

    rc = i = 0;

    /* read the first key */

    while ( ( rc = fread( &b[ i ] , 1 , 1 , fp ) ) != 0 )
        if ( b[ i++ ] == '\t' )
            break;

    if ( rc == 0 )
        {
        fclose( fp );
        return( 0 );
        }

    /* scan for requested key */

    if ( key->flags != MASK_FIND_LEFT )
        while ( 1 )
            {
            i = 0;
            while ( ( key->data[ i ] == b[ i ] ) && ( i < key->length ) )
                i++;
            if ( ( b[ i ] == '\t' ) && ( i == key->length ) )
                break;

            i = 0;
            while ( ( rc = fread( &b[ i ] , 1 , 1 , fp ) ) != 0 )
                if ( b[ i++ ] == '\n' )
                    break;
            i = 0;
            while ( ( rc = fread( &b[ i ] , 1 , 1 , fp ) ) != 0 )
                if ( b[ i++ ] == '\t' )
                    break;

            if ( rc == 0 )
                {
                fclose( fp );
                return( 0 );
```

```
        }
    }

    if ( key->flags == MASK_FIND_NEXT )
        {
        i = 0;
        while ( ( rc = fread( &b[ i ] , 1 , 1 , fp ) ) != 0 )
            if ( b[ i++ ] == '\n' )
                break;

        i = 0;
        while ( ( rc = fread( &b[ i ] , 1 , 1 , fp ) ) != 0 )
            if ( b[ i++ ] == '\t' )
                break;

        if ( rc == 0 )
            {
            fclose( fp );
            return( 0 );
            }
        }

    /* copy located key into return area */

    key->length = 0;
    while ( ( key->data[ key->length ] = b[ key->length ] ) != '\t' )
        key->length++;

    /* copy data part of record */

    buf->length = 0;
    while ( ( rc = fread( &buf->data[ buf->length ] , 1 , 1 , fp ) ) != 0 )
        if ( buf->data[ buf->length ] == '\n' )
            break;
        else
            buf->length++;

    fclose( fp );
    return( rc );
    } /* Find */


int Insert( key , buf , fn )
    struct  data_rec    *key;
    struct  data_rec    *buf;
    char                *fn;
    {
    FILE    *fp;
    char    b[ MAX_PARSE ];

    key->data[ key->length++ ] = '\t';
    key->data[ key->length ] = '\0';

    if ( ( key->flags & MASK_INS_REPLACE ) == 0 )
        {
        /* put the data into a temporary file */
```

```
                                    /* record key */
                                   /* record data */
                                     /* file name */

                                  /* file pointer */
                                    /* I/O buffer */
```



```
        buf->data[ buf->length++ ] = '\n';
        fp = fdopen( creat( "itmp" , 0600 ) , "w" );
        fwrite( key->data , 1 , key->length , fp );
        fwrite( buf->data , 1 , buf->length , fp );
        fclose( fp );

        /* combine the temporary and real files */

        (void) sprintf( b , "cat %s itmp > itmp2" , fn );
        system( b );
        system( "rm itmp" );
        }

    else
        {
        /* replace the record */

        (void) sprintf( b , "sed '/%s/s/%s.*/%s%s/' %s > itmp2" , key->data ,
                key->data , key->data , buf->data , fn );
        system( b );
        } /* if */

    /* make the sorted file the real file */

    (void) sprintf( b , "sort -n itmp2 > itmp3" , fn );
    system( b );
    (void) sprintf( b , "mv itmp3 %s" , fn );
    system( b );
    system( "rm itmp2" );

    key->data[ --key->length ] = '\0';
    return( 1 );

    } /* Insert */


int Delete( key , buf , fn )
    struct  data_rec    *key;
    struct  data_rec    *buf;
    char                *fn;
    {
    FILE    *fp;
    char    b[ MAX_PARSE ];
    int     i;

    /* check that file exists */

    if ( ( fp = fopen( fn , "r" ) ) == NULL )
        return( 0 );
    else
        fclose( fp );

    /* form key for scanning */

    i = 0;
```

```
                                    /* record key */
                                   /* record data */
                                     /* file name */

                                  /* file pointer */
                                    /* I/O buffer */
                                        /* index */
```

```
    while ( key->data[ i ] != '\0' )
        i++;

    key->data[ i++ ] = '\t';
    key->data[ i ] = '\0';
    buf->data[ buf->length ] = '\0';
    (void) strcat( key->data , buf->data );

    /* grep file for specified record */

    (void) sprintf( b , "grep -v '%s' %s > dtmp" , key->data , fn );
    system( b );

    /* rename updated file to real file */

    (void) sprintf( b , "mv dtmp %s" , fn );
    system( b );
    return( 1 );

    } /* Delete */


int Creat( fn )
    char *fn;                                   /* file name */
    {
    /* just reset the file */

    return( creat( fn , 0600 ) );
    } /* Creat */
```