The University of Alberta

Access Methods for a Semilattice Database Management System

by

© Ajit Singh

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, 1986

ISBN 0-315-30290-9

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Ajit Singh

TITLE OF THESIS: Access Methods for a Semilattice Database Management System

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1986

(Signed) [illegible]

Permanent Address:
c/o Dr. A. D. Taskar
F-Block, 5B Saket,
New Delhi 110 017, INDIA.

Dated January 23, 1986

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 Access Methods for a Semilattice Database Management System submitted by Ajit Singh in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]
....
Dr. William W. Armstrong, supervisor

[illegible]
....
Dr. Mark Green

[illegible]
....
Dr. Stan Cabay

Peter R. Winters
....
Dr. Peter Winters

Date December 5, 1985

To my father

ABSTRACT

The semilattice data model is a new model proposed by W. W. Armstrong. The model has the ability to represent complex objects at a conceptual level through a non-normalized relational model allowing formation of tuples, sets, sequences, and union types. Internally its use of special purpose codes, called semilattice codes, aims at low data redundancy and attempts to reduce the need for disc accesses by embedding information about values referred to in the codes referring to them. A semilattice database management system (SL-DBMS) based on this model is under development at the Department of Computing Science, University of Alberta. This thesis investigates the requirements of the physical level of the SL-DBMS. A single storage structure is proposed for the physical organization of data in a SL-DBMS, called the C-B^*-tree. Three forms of this structure, called the encoding C-B^*-tree, decoding C-B^*-tree and secondary C-B^*-tree can be used for all kinds of operations in a SL-DBMS.

An algorithm for handling page-overflows and underflows, known as a pagination strategy, has been developed. This pagination strategy is also applicable to other variants of variable-length-record B-tree structures. Experiments indicate that this strategy performs better than other pagination strategies so far known.

A feasibility study is done for extending the proposed physical model into an adaptive data structuring system. In such a system all the structuring of data is not specified or performed at the initial data load time, rather data gets organized in a more and more efficient manner as the usage patterns on data become clearer.

Acknowledgements

I would like to thank my supervisor, Dr. W. W. Armstrong, for his valuable advice, for making himself available for discussion in spite of his busy schedule, and his encouragement and criticism at each stage of my research. The quality and readability of this thesis depends to a large extent on his careful and patient reading of the rought draft of the thesis. I would also like to express my gratitude for his financial support.

Thanks are due to the members of my examining committee, Dr. Mark Green, Dr. Stanley Cabay and Dr. Peter Winters, for their helpful suggestions and comments. I would also like to acknowledge the Teaching and Research Assistantships provided by the Department of Computing Science.

Most of all, I would like to thank my family and friends, for their support and encouragement throughout my academic career.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. The Problem

A data model, in general consists of two elements : a mathematical notation for expressing data and relationships, and operations on data that serve to express queries and other manipulations of data. The three well known data models, namely, relational, network, and hierarchical, have been used in the great bulk of commercial database systems. The semilattice data model is a new data model proposed by W.W. Armstrong [Arm84]. It has the ability to represent complex objects at a conceptual level through a non-normalized relational model allowing formation of tuples, sets, sequences and union types. Internally, its use of special purpose codes, called semilattice codes, aims at low redundancy and attempts to reduce the need for disk accesses by embedding information about values referred to in the codes referring to them. These are the two most important features of the model that make it different from other data models. Both the conceptual and internal models are important, since the conceptual model needs the internal model to overcome problems of non-normalization. Like Codd's relational data model, the semilattice conceptual non-normalized relational data model also is divorced from various implementation considerations (in order to provide a high degree of data independence). It provides the user with a very high level and entirely non-procedural facility for data definition, retrieval, update, access control, support of views, and verification of various kinds of constraints.

To a large extent, the acceptance and value of the semilattice data model's approach hinges on the demonstration that a system can be built which can be used in a real environment to provide a solution to a certain class of problems faced by the database user community. The implementation of a prototype database management system based upon the semilattice data model is currently in progress at the Department of Computing Science, University of Alberta. The purpose of this thesis is to provide a description of the overall architecture of the proposed physical data model for the semilattice data management system.

1.2. An Outline of the Proposed Physical Data Model

At the heart of the proposed physical model for the semilattice DBMS is a data structure called C-B^*-tree. The C-B^*-tree structure is a variant of the well known B-tree structure for indexed files on secondary storage [BaM72]. In general B-tree structures have the following features:

(1) Utilization of Secondary storage space better than 66% is maintained at all times. Storage space is dynamically allocated and reclaimed, and no service degradation occurs when storage utilization becomes very high.

(2) Random access requires very few physical block accesses and is comparable to other random access methods.

(3) Efficient sequential processing of records is also allowed.

(4) Record deletions and insertions are efficiently handled usually keeping the modification of the structure localized. Updates allow maintenance of the natural order of keys for sequential processing and proper tree balance for fast random retrieval.

In addition to these, the proposed C-B^*-tree structure has the following features:

(1) The structure is designed to support retrieval on both primary and secondary indexes. Also, combined indexing on a particular kind of data, namely the semilattice codes, is possible.

(2). A compression scheme is employed that does not repeat a key common to many records that reside on the same page. There is no limit to the number of records that may have the same key and records having the same key may be distributed over an arbitrary number of pages.

(3) Variable-length records and keys are allowed. This provides a great amount of flexibility in the representation of data.

(4) A modified algorithm, known as a pagination strategy, is used for handling the distribution of records among neighboring pages in the event of page overflow or page underflow. It has better performance than other proposed pagination strategies for variable-length-record B-tree structures.

The Adaptive Component

A preliminary study is done for extending the above mentioned physical model into what we call an Adaptive Data Structuring System (ADSS). In ADSS all the structuring of data is not done at the initial data load time, rather the data gets more and more efficiently organized as the usage patterns on data become clearer. The various steps during the organization of data by ADSS are shown in figure 1.1.

The first step involves the specification of certain data organizations by the user or the database administrator and initial loading of data according to these schemes. The second step entails the monitoring of actual system operations. The monitoring task provides information about more frequent usage patterns on the data. This may help in identifying user requirements and critical performance areas. During the next step the monitored data is analyzed and decisions regarding modified organizations for data are taken. The final step addresses the changes in the data structures and tuning (within the various degrees of freedom available) of the current structures on the basis of feedback received from the previous step and the consent of the database administrator. The objective is to optimize performance within the existing system by reorganizing the database according to the changing interests of its user community.

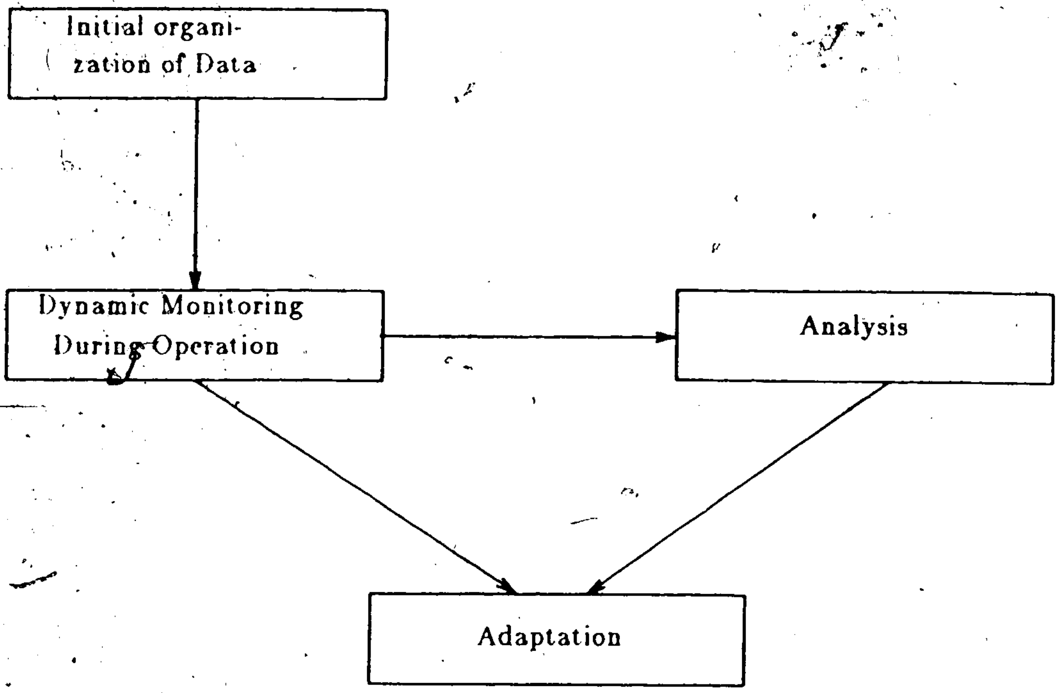
Initial organization of Data
Dynamic Monitoring During Operation
Analysis
Adaptation

Figure 1.1 Adaptive Organization of Data

1.3. Organization of the Thesis

Chapter 2 of this thesis presents a review of the background material required for our subject. A brief review of the semilattice data model is presented. Two example semilattice database schemas are given. An instance of a database based upon each of these schemas is given in Appendix A. These example schemas are referred to throughout the thesis to illustrate various points regarding the semilattice data model. Next, we take a brief look at various data structures that are frequently used by database systems and current trends in this area. Finally, a short review of the adaptive techniques so far used in conjunction with database management systems is presented.

A storage structure, called the C-B^*-tree and its associated access mechanisms are presented in chapter 3. The C-B^*-tree structure is used for storing relations and other kinds of information associated with a semilattice database system. The proposed storage structure and its access mechanisms are analyzed in the light of the requirements of a semilattice database system. Some experiments are performed in order to arrive at a suitable pagination strategy for the C-B^*-tree. The theme of these experiments is explained and their results are analyzed. A new pagination strategy, called CS strategy is presented for variants of variable-length-record B-trees, including the C-B^*-tree.

The purpose of Chapter 4 is to describe the overall architecture of the proposed adaptive physical model. Objectives of adaptive data organization are explained. Various components of the Adaptive data structuring system are ~~briefly~~ described. Two important areas that need to be further looked into are identified. These are : choice of secondary indexes, and attainment of better efficiency through introducing controlled redundancy in the database. The issues that affect each of these decisions are analyzed. Emphasis is on aspects that require more work in order to make the proposed ADSS complete. The last chapter of the thesis discusses the areas that need further work.

1.4. Indication of Original Contribution

The work in this thesis is a part of the larger effort that is currently being made in order to develop the first prototype DBMS based on the semilattice data model [Bob84, Arm84]. About a year of contemplation and discussions convinced us of the practical feasibility of the model. This thesis clearly demonstrates this fact for the physical organization of data and associated basic operations necessary for a semilattice database system. The description of the proposed model for this purpose is the most important achievement of this thesis. A single storage structure which can be efficiently used for all kinds of query and update operations on a semilattice database system is proposed. The main body of this proposed structure has been implemented under the 4.2 BSD UNIX† operating system at the Department of Computing Science,

† UNIX is a trademark of AT&T Bell Laboratories.

University of Alberta. The implementation consists of over 5000 lines of source code written in the C language.

A pagination strategy for the variable length record B-tree structure is proposed. Experiments have shown that this strategy performs better than other pagination algorithms currently known. This came as an offshoot of our main work of designing a special kind of variable length B-tree structure, called the C-B^*-tree for use with a semilattice database system. In this context a new performance measure called **data capacity** for evaluating a pagination strategy is proposed.

The overall architecture of an adaptive data structuring system is described which can be used in conjunction with the above mentioned storage structure. An indication of future work needed to complete the design of the ADSS is given.

Chapter 2

A Review of Background Material

2.1. Introduction

The subjects chosen for review in this Chapter fall into three areas. First, a review of the salient features of the semilattice data model is presented. For this, we shall limit ourselves to two aspects of the model that are of chief concern for the design of the physical data model. These aspects are : the ability of the semilattice model to represent large and complex objects in a structured manner and its use of semi-meaningful internal codes in place of keys or physical pointers. Next, a bird's eye view of the various kinds of data structures that are often employed by databases for physical organization of data is presented. Lastly, a review of past and current work in the area of adaptive physical organization of data is given.

2.2. The Semilattice Model

Recently, many researchers have pointed towards the inadequacies of the normalized relational system for purposes such as engineering design applications and graphics. A number of non-normalized relational systems have been described to improve the situation [Har84, HNC84, AbB84, KKA84]. The semilattice data model also is a non-normalized relational data model which attempts to overcome the difficulties with normalized relational systems. It uses the relational data model in such a way that large, complex objects can be effectively stored, retrieved and manipulated.

According to the first normal form of the relational data model a relation is allowed to have only atomic values in its domains. In the semilattice data model, a domain is allowed to be an atomic value, a tuple, a sequence or a set of values. Also, the type of value assigned to an attribute may be one of a given set of alternative types (i.e. it is a union-type). The actual type of data is then indicated by a data descriptor or tag value. One component of a disjoint union is allowed to have a null tag. This means that if data is encountered which has no tag value, it is considered to have a null tag. This is used for elaborating a database scheme without having to reload all existing data of a certain type with a tag value.

2.2.1. Basic Definitions

Identifier

An identifier is a character string naming a type or a value. It is formed according to a certain set of rules.

Type

A type is a set of values plus an associated set of operations on these values. The set of values are called the domain of the type.

Types are either basic (or atomic) or constructed. The basic types are: integer, real, string of characters and boolean with the usual domains and operations.

Type Expression

The type of a data is denoted by a type expression. A type expression is either a basic type or is formed by applying an operator called type constructor to other type expressions. The available type constructors are: set-scheme, sequence-scheme, tuple-scheme and disjoint-union-scheme. These type constructors are denoted by structures like $\{t_1\}$, $<t_1>$, $(t_1, t_2, \cdots t_n)$ and $(t_1 \mid t_2 \mid \ldots \mid t_n)$ respectively where t_i s denote already defined types.

For example, suppose the set of all nonempty strings is named "string". Then {string} represents the type whose domain is the set of all sets of strings and <string> denotes the type whose domain is the set of all finite sequences of strings. Similarly, (symbols:{string},names:<string>) represents the tuple-scheme in which the domain of attribute 'symbols' is the set of all sets of strings and the domain of 'names' is the set of all finite sequences of strings. An example of disjoint-union-scheme is the type (symbols:{string} | names:<string>). A value of this type would consist of the tag 'symbols' or 'names', and a value belonging to the domain of 'symbols' or 'names' respectively.

A type expression may be named using an identifier followed by an equal sign ('=') followed by the type expression. Therefore, a type name is also a type expression.

The above definition of type is quite in accordance with the definition of data type given in the literature related to programming languages [Ten81, ASU86]. We shall not describe here the operations that are defined for a type. This has been discussed, to some extent, in Armstrong's original paper [Arm84] and further research on this subject is continuing.

Variable

A variable is an identifier which can be assigned a value from a domain.

Constraints

Constraints prescribe allowable values for a variable.

Attribute

An attribute is an identifier that selects a component of a tuple_scheme.

Tuple

A tuple is a value constructed according to a tuple_scheme.

Relation

A relation is a value and is a set of tuples formed according to a given tuple_scheme.

Given that the value assigned to an attribute can be arbitrarily complex, a tuple can be an arbitrarily complex object, and can even represent an entire relational database. In the semilattice model, the tuple_scheme rather than relation scheme or database scheme may be regarded as fundamental. However, for the sake of denotation, a relation name in Codd's original model is here represented by the same name as the tuple-scheme but written in upper-case letters. If two or more relations use the same tuple_scheme, we must find an alternative notation for their names. A database scheme then is a collection of relation schemes with some inter-relational constraints. The generalized form of the definition of domain allows us to capture the essence of a database scheme by using a tuple_scheme whose attributes are the relation names (including constraints which are then on the allowable tuples, i.e. database instances). In this way a database instance in Codd's model becomes a single tuple in the semilattice model. The values of the attributes are relations.

2.2.2. Some Examples of Semilattice Database Schemes

In this section we present three examples of database schemes constructed according to the semilattice data model. These examples illustrate the application of the semilattice data model in three quite different areas of database application. However, a reader may skip reading the last two examples without any significant loss in understanding of the rest of the material in this thesis.

(a) The Chip Database

Here is an example of a database scheme which describes functional diagrams (or stick diagrams) of simple electronic circuits. A database scheme for this database in the semilattice data model is shown in figure 2.1.

```
point   = (x :real, y :real)
POINT   : {point}
line    = (point1 :point, point2 :point)
LINE    : {line}
polygon = (shape  : {line})
POLYGON : {polygon}
```

Constraints:

(1) Values of *point1* and *point2* in relation LINE are members of POINT.

(2) Value of each member of *shape* in a tuple of POLYGON is a member of LINE.

(3) The set of lines in each tuple of relation POLYGON describe a closed polygon.

```
block    = (ent :entity, instance :integer, place :point)
BLOCK    : {block}
```

Constraints:

(1) The value of *place* is a member of POINT.

(2) The value of *ent* is a member of ENTITY.

```
pin      = (ent :entity, class :binary, pin# :integer,
           place :point)
PIN      : {pin}
```

Constraints:

(1) The value of *place* in each tuple of PIN is a member of POINT.

(2) The value of *ent* is a member of ENTITY.

```
connection = (block1 :block, pin1 :pin, block2 :block,
        pin2 :pin, con_seg :<line>)
CONNECTION : {connection}
```

Constraints:

(1) The values of *block1* and *block2* in each tuple of CONNECTION are each members of BLOCK.

(2) The values of *pin1* and *pin2* in each tuple of CONNECTION are each members of PIN.

(3) In the value of *con_seg* each member of the sequence is a member of LINE.

(4) In the sequence of lines contained in each value of the attribute *con_seg*, the end point of a line is the beginning point of the following line.

```
bas_entity   = (entid :integer, shape :polygon,
                    ports :{pin})
BAS_ENTITY   : {bas_entity}
```

Constraints:

(1) The value of *shape* is a member of POLYGON.

(2) Each member of the value of *ports* is a member of PIN.

```
com_entity   = (entid :integer, shape :polygon,
     components : {block}, ports :{pin},
     inter_con :{connection})
COM_ENTITY   : {com_entity}
```

Constraints:

(1) The value of *shape* is a member of POLYGON.

(2) Each member of the value of *ports* is a member of PIN.

(3) Each member of the value of *component* is a member of BLOCK.

(4) Each member of the value of *component* is a member of CONNECTION.

(5) An entity can not have itself as a component i.e. *entid* of a COM_ENTITY should be different from *entid* of each of its component block.

```
entity = (b_ent:bas_entity|c_ent:com_entity)
ENTITY : {entity}
```

Constraint:

The value of *ent* is a member of either BAS_ENTITY or COM_ENTITY.

Figure 2.1 Tuple_schemes and Relations for the Chip Database

The relations POINT and LINE represent all the points and lines respectively in the diagram that are of interest. The relation POLYGON describes the exterior shape of each component in the circuit. These shapes are closed polygons in general. The relation BLOCK lists various instances of entities that are used as subcomponents in some other entity. Various input and output pins are described by the relation PIN. The relation CONNECTION describes the pin connections among various blocks. The attribute *con_seg* represents a sequence of lines that describe the connecting route between the two given pins. The relation BAS_ENTITY describes basic cells of the chip (2-input AND gate in the present case) whereas the relation COM_ENTITY describes compound cells i.e. cells that have other cells as their components. The relation ENTITY then gives the overall description of an entity in terms of its geometry, subcomponents, and their interconnections. It is expressed as a disjoint union of the BAS_ENTITY and the COM_ENTITY relations. It is worth noting that no attempt need be made to refer to entities by a particular key. In principle, any one of several possible keys could be used. The semilattice conceptual model does not require a choice.

(b) The HVFC Database

Our next example of a database scheme is based on the example of Happy Valley Food Cooperative (HVFC) database of Ullman[Ull82]. The scheme of the HVFC database in the semilattice data model is given in figure 2.2. The relation MEMBER represents the members of the HVFC and their account balances. Various items that are supplied to the HVFC or those sold to its members by HVFC along with their prices are given by the relation PRICED_ITEM. A tuple of the ORDER relation describes an item ordered by a member. Each tuple of the relation SUPPLIER describes the set of items that are possibly supplied by a certain supplier.

```
member     = (name :char(30), address :char(50),
              balance :integer)
MEMBER     : {member}

priced_item = (item: char(20), price: integer)
PRICED_ITEM : {priced_item}

order      = (order_no :integer, purchaser :member,
              item : priced_item, quantity :integer)
ORDER      : {order}
```

Constraints:

(1) The value of *purchaser* is a member of MEMBER.

(2) The value of *item* is a member of PRICED_ITEM.

```
supplier   = (sname :char(20), address :char(50),
              stock :{priced_item})
SUPPLIER   : {supplier}
```

Constraint:

The value of *stock* is a subset of PRICED_ITEM.

```
hvfc    = (members :{member},goods :{priced_item},
           orders :{order}, suppliers :{supplier})
```

Figure 2.2 Tuple_schemes and Relations for the HVFC Database

(c) The Graphics Database

The third example illustrates (figure 2.3) the specification as well as usefulness of the disjoint union scheme for domains. It is taken from [Arm84]. An object consists of either a square or a rectangle. A picture is a set of such objects.

```
point  = (x : real, y : real)
POINT  : {point}

circle = (centre : point, radius : real)
CIRCLE : {circle}
```

Constraint:

The value of *centre* is a member of POINT.

```
rectangle  = (lower_left : point, upper_right : point)
RECTANGLE  : {rectangle}
```

Constraints:

(1) The value of *lower_left* is a member of POINT.

(2) The value of *upper_right* is a member of POINT.

(3) The values of x and y of *lower_left* are less than values of x and y respectively of *upper_right*.

```
object  = (obj# : (circle|rectangle))
OBJECT  : {object}
```

Constraint:

The value of *obj#* is a member of either CIRCLE or RECTANGLE.

```
picture  = (pic# : {object})
PICTURE  : {picture}
```

Constraint:

The value of *pic#* is a member of OBJECT.

Figure 2.3 Tuple_schemes and Relations for the Graphics Database

2.2.3. The Semilattice Condition

Before we can describe the mechanism of referencing in the semilattice ~~model,~~ it is necessary to explain what is meant by the semilattice of a database. A directed graph can be formed from variable names by using arcs whereby the variable at the upper end of the arc will, by convention, have as value an object which is a part of the object which is the value of the variable at the lower end. In some cases, the object at the lower end of the arc can be defined in terms of the objects at the upper ends of arcs coming into its scheme either as a union or as a join. Even if the relation at the lower end of several arcs cannot be defined as a join or union in this way, it indicates that the object at the lower end refers to the values at the upper ends.

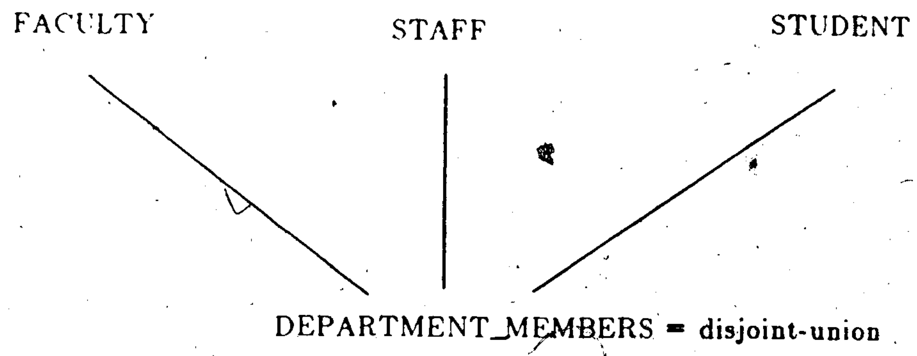

Figure 2.4 Definition of a Relation Using Disjoint-union Scheme

For example, the relation DEPARTMENT_MEMBERS (figure 2.4) is a disjoint union of the relations FACULTY, STAFF, and STUDENTS. The relation employees in figure 2.5 is a union of three smaller employee relations that represent employees at different locations. Figure 2.6 illustrates the formation of a relation that is not defined in terms of relations at the upper level rather, the relation SUPPLIER_PART_QUANTITY refers to some of the tuples of SUPPLIER and PART relations.

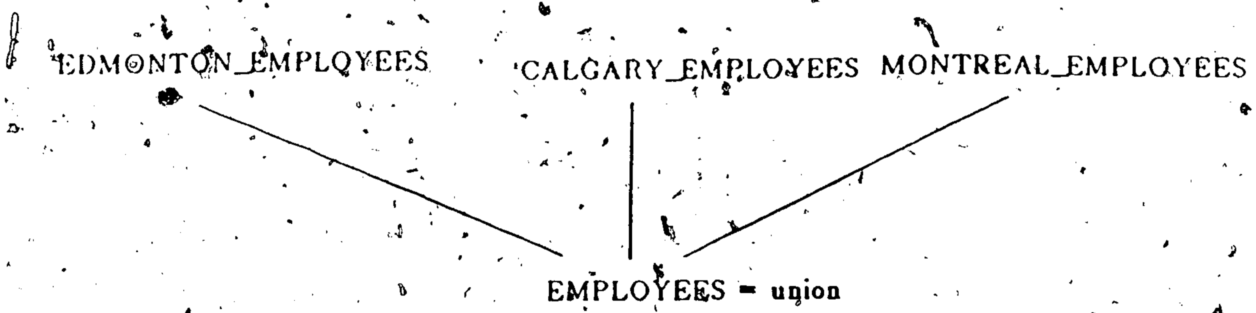

Figure 2.5 Definition of a Relation as a Union of Relations

In general, the semilattice condition for forming a diagram is the following: if the values of two variables have a common part (e.g. relations that 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. The situation with subsets is analogous: if two relation variables R1, R2 correspond to relations having a non-empty intersection, then their intersection R3 need only be stored once. The intersecting part of a relation would be stored as a reference to the relation of R3 (figure 2.7).

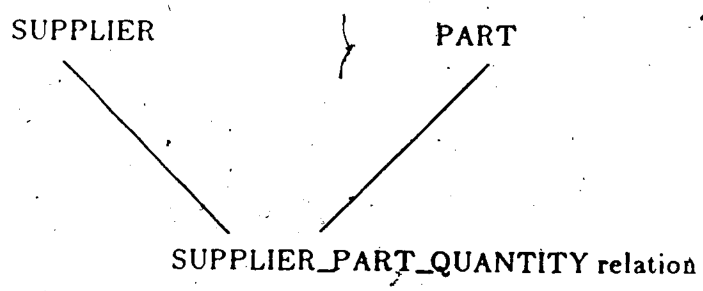

Figure 2.6 A Relation Referring to T[illegible]es in Other Relations

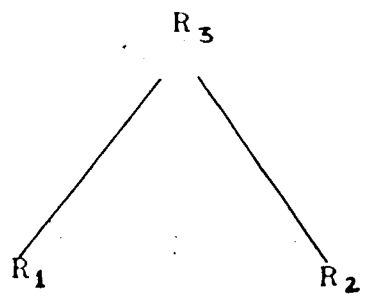

Figure 2.7 Referring to a Common Part

2.2.4. Internal Structure of a Semilattice Database

Brief codes like small integer numbers can be used internally to identify tuples in the smallest part-relations. Codes for tuples in a relation which contain other tuples as values of attributes can be obtained by concatenating the codes defined for the projections with the assigned code for the uncoded part of the tuple. For example, in the instance of the Chip database given in Appendix A, codes for tuples in relation LINE are obtained by concatenating the corresponding codes for *point1* and *point2* respectively. Codes for the parts of a union require a type discriminator, concatenated with the code for the member within its part of the union. An example of this is the ENTITY relation of the Chip database where the null tag is used to indicate that the tuple referred to belongs to the BAS_ENTITY relation and a tag value 1 is used to indicate that it belongs to the COM_ENTITY relation.

One method of assigning a code for the uncoded part of the tuple is to assign a relative code with respect to the already coded part of the tuple. The code for the complete tuple then consists of a combination of codes for all its parts concatenated in a certain fixed order. This method of coding is called relative coding. For example, codes for tuples in the ORDER relation of the HVFC database are generated by this method; the complete code for a tuple consists of a generated relative code for *order-no* and *quantity* concatenated with codes for *purchaser* and *item*. Another method of assigning codes to the tuples in a relation would be to assign fixed-length, unique identifiers to tuples. This scheme of code generation is called absolute coding. For example, this method is used for coding tuples of the SUPPLIER relation of the HVFC database. In the relative coding scheme a code for a tuple contains more information about the parts of the tuple it represents. This might result in significant improvement in the efficiency of a database system because some processing may be done on codes rather than actual attribute values. In the absolute coding scheme the amount of processing that can be done on codes only would be much less because of the lack of knowledge about the values of the constituents of the tuple. However, the relative coding scheme might result in arbitrarily long codes for the tuples which have non-atomic domains like sets, sequences etc. in their tuple-schemes. Also, the process of generating codes would require operations like comparing two complex structures of codes (e.g. two sets of codes) for equality where the members of such a structure can in turn be structures of some other kind and so on.

Although the semilattice data model does not involve itself in specifying the implementation-related details, we here propose the following convention for generation of codes for the tuples in a semilattice database system:

A relative coding scheme is followed for coding the tuples of a relation whose tuple-scheme does not have sets or sequences as members of the domain of any of its attributes. An absolute coding scheme is used for tuple-schemes that have at least one such domain in their tuple-schemes.

An instance of a database based on the scheme of the Chip database is presented in Appendix-A, section A-1. Attributes or tuples whose values are in the coded form are indicated with the presence of a slash(/) character. For example, A/ indicates absolute code for A whereas A/B indicates code for A relative to B. For the reasons explained above, the tuples in POLYGON, CONNECTION, BAS_ENTITY and

COM_ENTITY relations are coded according to the absolute coding scheme whereas the other five relations in the database scheme are coded according to the relative coding scheme. There is no referred to part in the tuple-schemes for relations POINT and LINE. Therefore, the absolute coding for their tuples would be the same as the relative coding. Section A-2 of Appendix-A contains an instance of a database based on the HVFC database scheme. Again, relative codes are assigned to the tuples of the MEMBER, ORDER and PRICED_ITEM relations whereas an absolute code is assigned to each tuple in the SUPPLIER relation. There is no relation corresponding to the tuple-scheme **hvfc** since such a relation can be constructed from the other four stored relations.

One of the key points in the setup of an internal structure for a SLDB is that because codes are used, the storage space required for building a new secondary index for a relation is small compared to what would be required if actual values are used. The second key point is that, again because codes are used, more useful data can, in principle, be obtained per disc access. With proper clustering of data this should be beneficial for performance.

2.2.5. Operations in the Semilattice Data Model

Operations of the relational data model like selection, join, projection, cartesian product, union, difference etc. have their analogs in the semilattice data model. These operations have been defined keeping in mind the non-normalized structure of tuple-schemes in the semilattice data model. Instead of repeating the description of these operations the reader is referred to the original paper [Arm84].

The next section presents an overview of various data structures generally employed in physical data models of various database management systems.

2.3. Classification of Data Structures for Database Management Systems

The design of data structures for a DBMS directly contributes to the usability and responsiveness of a database system. The initial data structures used for databases were sequential in nature - a sequential deck of cards or its image on magnetic tape. Insertions and deletions in this kind of data structure lead to at least one of two undesirable consequences: the introduction of ad-hoc mechanisms (such as a flag to indicate that a record still present should be considered as having been deleted, or pointers to an overflow bucket which holds records that can not be squeezed into their rightful place), and frequent expensive restructuring of the entire data (typically when the number of holes left by deletions, and overflow areas created by insertions, have grown so large as to severely degrade performance).

Direct access storage devices such as drum and disk made organizations such as ISAM [IBM66] possible. These access methods usually permit access to each record in two steps: first a directory is searched which points to the proper cylinder or track; second, this track is searched sequentially. For static files this scheme is as fast as the hardware restrictions on disk accessing permit. For highly dynamic files index sequential structure could lead to very poor performance; instead of the two-step access to data, long linear chains of "overflow buckets" may be traversed.

Balanced trees turned out to be a good solution for storing highly dynamic files on secondary storage. The B-trees of Bayer and McCrieght [BaM72] were the first file organization schemes that addressed together the issues of storage efficiency and dynamic adaptation of structure to fit data. A number of variants of B-tree structure have been proposed since then [Knu73,BaU77,McC77].

Multidimensional tree structures have been proposed which enable access to a file on more than one attribute. Notable among these are the multidimensional binary tree structure of Bentley [Ben75], multi-attribute tree (MAT) of Chang and Fu [ChF79] and K-D-B-trees of Robinson [Rob81]. Although these structures offer the flexibility of access to more than one attribute, a common shortcoming of these methods is that they do not fare well in case of dynamic databases. For example, k-d trees of Bentley require that database be "clustered" in a preprocessing step. Similarly, reorganizational algorithms for K-D-B-trees have not yet been fully developed [Rob81].

Another kind of data structure which was based on key-to-address transformation, called hashing, was discussed in 1969, giving O(1) access time to individual records in a file [Ols69]. One of the major drawbacks of hashing schemes is the static storage allocation i.e. the size of the file must be estimated in advance and storage space must be allocated for the whole file. With dynamic files the performance of hashing schemes may become very bad. It may even become necessary to rehash all the records into a new file. Also, if the hashing scheme is not order-preserving it becomes unsuitable for answering range queries. Various kinds of hashing techniques are discussed by Sorenson et al. [Sor78].

Hashing schemes for dynamic files have been proposed by Knott [Kno71], Larson [Lar80], Litwin [Lit80] and Fagin et al. [FNP79]. A multidimensional hashing method for static files was proposed by Rothnie and Lozano [RoL74]. Another order-preserving multidimensional hashing method for static files has been proposed by Merrett and Otoo [MeO78].

Parallel to the development of the above mentioned data structures, several other less frequently discussed and used data structures were developed. Some of the important ones among them are multilist, controlled list-length multilist and cellular data structures [TrS76].

The work on data structures for databases can be classified in several ways. First, there is a distinction between direct storage organizations [Ols69, Kno71, Lar80, Lit80, FNP79, RoL74, MeO78] and tree structured organizations [Dum56, IBM66, BaM72, Ben75, FiB75, LeW80, LeY77, ChF79, Rob81]. This is a distinction between O(1) access time and O(log n) access time. Second we can classify these data structures as single attribute access [Dum56, BaM72, Ols69, Kno71, Lar80, Lit80, FNP79] and multi-attribute access [Ben75, FiB75, LeW80, LeY77, ChF79, Rob81, RoL74, MeO78]. The third distinction is between static [Dum56, MeO78, OlS69, RoL74] and dynamic organizations [BaM72, Ben75, FiB75, LeW80, LeY77, ChF79, Rob81, Kno71, Lar80, Lit80, FNP79]. Lastly, order-preserving methods [BaM72, BeN75, FiB75, LeW80, LeY77, ChF79, Rob81, MeO78] are distinguished from the others [Dum56, Ols69, Kno71, Lar80, Lit80, FNP79, Rol74].

2.3.1. Selecting Data Structure for a DBMS

On the basis of the above-mentioned distinctions it is apparent that a storage-efficient, dynamic, order-preserving, multi-attribute direct access storage organization would be an ideal choice for a DBMS. Unfortunately none of the so far known data organization schemes fulfills all these requirements. Therefore, one of the most difficult decisions the implementors of a DBMS should take is the selection of data structures and access methods that should be provided with the DBMS. A survey of access methods used in recent database management systems, however, does indicate a few interesting and definite trends.

First, most of the systems offer more than one option to the database administrator for organization of data to suit different types of applications. For example, the DBTG group recommended five options, called "location modes", to the DBA [Wie77]. The implementors of INGRES provide three options for the organization of relations [SWK76, Sto80]. These options are sequential (heap), hashed, and indexed sequential organizations. Both primary and secondary indexes are supported.

Second, the designers of most systems have elected to implement a sequential structure (i.e. a non-keyed structure) along with one or two keyed file structures. Keyed file structures which are more frequently chosen are hashed, indexed sequential and inverted file structures. The multilist and cellular partitioned structures are less frequently used.

Third, earlier systems placed a lot of emphasis on storage space (RM/XRM [Lor74], PRTV [Tod76, Tod77], RDMS [StG74]). Multiple-byte character strings were stored in an indirect manner. First, they were represented by numerical fixed length identifiers which were later used as references to the original data. Possibly because of rapid decline in memory costs, this division of memory space no longer seems fashionable. In recent systems there is a trend to store variable-length character strings directly (INGRES [Sto76, Sto80], SYSTEM R [Ast76, Bla81, Cha81] and ADABAS [Atr80]).

Fourth, very few systems support some kind of compression/decompression scheme in order to increase storage space utilization. PRTV and INGRES are two such relational systems. INGRES supports data compression as an option. A major drawback of such schemes is the high processing overhead by data compression/decompression routines (20% of CPU time in case of PRTV).

2.4. Physical Design Aids for Organization of Data

One of the most difficult tasks a database administrator(DBA) must perform is the selection and periodic evaluation of access methods which would provide satisfactory, if not optimal, performance of all applications. Although the decision of the DBA in most cases is still based on his intuitive and qualitative impressions, there are some simulation and analytic models available to assist in this task.

The principal origin of these works can be traced to the formal model for the list oriented data structures developed by Hsiao and Harary [HsH70]. This model views the file as a collection of records organized on linked lists. A directory is used to model the access to the records by indicating the record key values, the number of records containing these keys, the number of lists each key is on, and a pointer to each list. Using a simulation approach Cardenas built a system to aid in the selection of various file structures. The selection is based on a set of user requirements and a given set of cost equations for each file structure [Car75].

Severance, recognizing that the Hsiao-Harary model only addressed list type record sequencing, extended the model to include a broader class of structures [Sev72]. He introduced a two parameter (each two-valued) model to describe physical record accessing mechanisms- data direct, data indirect, pointer sequential, and address sequential. Given the characterization of the database and a series of cost equations, Severance developed a semi-heuristic search mechanism to identify the optimal design.

In a major theoretical breakthrough, Yao formulated an analytic approach to modeling storage structures in which a single model and cost function could be used to characterize most storage organization alternatives [YaM75, Yao77]. Sequential, indexed sequential, hashed, multilist, and inverted organizations could all be defined in terms of the basic parameters of the general model.

The theoretical work of Yao was extended and implemented to characterize existing storage organizations in existing environments. A software package called the File Design Analyzer (FDA) was developed to evaluate well-known file organizations in terms of I/O processing time and secondary storage space required to service a set of user applications [TeD76].

2.5. Monitoring, Analysis and Adaptation of Physical Organisations

Monitoring is perhaps the most useful tool in the design of storage structures for databases. A monitoring package is needed to collect statistics on the utilization of the database access paths. Based upon the inputs from this phase or changes in the basic requirements it might be necessary to modify database structures. Here we shall limit our discussion to modifications in the physical organization of data to the various possible degrees of freedom. Belford at the University of Illinois, proposed a monitoring mechanism based upon well-known statistical gathering procedures [Bel75]. In this preliminary study algorithms were investigated to collect statistics on data usage patterns exhibited by the application programs. They are currently developing tests and simulations to evaluate and determine the effect of various parameter choices on the efficiency of the algorithms.

Two of the few reported implementations of DBMS monitors are by Krinos [Kri73] and by Oliver and Joyce[OlJ76]. Krinos describes a monitor implemented for the United Aircraft Information System (UAIMS) which compares the activities of DBMS applications with non-DBMS applications. Oliver and Joyce report on their experiences with REGIS, the Relational General Information System. Their performance monitor collected data about utilization of the REGIS command language.

Some of the works that address the issue of dynamic reorganization of data based upon usage pattern analysis on data are described by C. T. Yu et al., Hammer and Niamir, and Hammer and Chan. Hammer and Chan proposed a heuristic solution to the problem of selecting attributes on which secondary indexes should be created [HaC76]. Hammer and Niamir present a heuristic to divide a large file with many attributes into small files with fewer attributes [HaN79]. The work of C. T. Yu et al. deals with an adaptive algorithm to cluster records into blocks for higher efficiency [Yu84]. Their algorithm is capable of detecting changes in user's access patterns and then suggesting an appropriate assignment of records to blocks.

2.6. Conclusions

The semilattice data model puts some specific requirements on the physical model for a semilattice DBMS which are different from any other model. This is due to the presence of non-atomic domains in tuple-schemes of a semilattice database system and the use of the semilattice codes. Although there is a large variety of known data structures to choose from, there is little to help us in making a good choice. Most of the physical design aids make the impractical assumption of having the prior knowledge of all of the applications for which a given database system would be used. Also, the practice of offering a myriad of data structures to a database administrator can at best be described as a compromise strategy.

The following chapter describes a data structure which offers a great amount of flexibility in storage of data. The fourth chapter indicates how this data structure can be exploited to dynamically adapt the physical organization of data in a semilattice database system.

Chapter 3

Storage Structure and Access Methods for a SLDB System

3.1. Introduction

The design of storage structures and access mechanisms has a major impact on database system performance because it is at this level that actual implementation takes place in physical storage. In the previous chapter, the basic features of the semilattice data model were discussed. This chapter analyzes the storage structure requirements of the data model. A general format of stored records in a semilattice database system is described. A new variant of B-tree called compressed-B^*-tree or C-B^*-tree is proposed for the organization of stored records into files. The results of some experiments done regarding the pagination strategy of the variable-length B-tree structure are presented and their implications are analyzed. Finally, a new pagination strategy called Combined Strategy(CS) for variable-length-record B-trees is discussed. Experimental results show that CS could result in significant performance improvement over the existing strategies, namely Equal Strategy (ES) and Minimum Strategy (MS) [McC77].

3.2. Requirements Analysis

As discussed in the previous chapter, in the semilattice data model, an attribute is capable of representing a structured object. It allows non-atomic structures like sets, disjoint union of domains etc. to be members of the domain of an attribute. Also, very often, the value of an attribute in a relation is obtained by referring to some other tuple(s) in some other relation and this reference is done with the help of logical codes called 'semilattice codes'. These features put some special requirements on the storage structure schemes and their associated access mechanisms for a semilattice database system. In order to analyze these requirements we shall very frequently refer to the two semilattice database examples that were introduced in the previous chapter:

the Chip database and the HVFC database.

3.2.1. Coding and Decoding

As described in the previous chapter, the physical representation of a tuple in a semilattice database generally consists of two parts : a coded part, and an uncoded part. The coded part consists of semilattice codes that represent values of attributes belonging to tuples in other relations plus a relative code that is generated to represent the uncoded part of the tuple. The uncoded part of a tuple consists of actual (user-supplied) values of attributes.

This way of representing a tuple suggests that during various kinds of query or update operations, codes would be used quite frequently instead of actual values. For example, suppose we want to add a new tuple to the MEMBER relation of the HVFC database. User supplied values of *name* and *balance* would be inserted into the MEMBER relation and a code would be generated for the newly inserted tuple. Deleting a tuple from the MEMBER relation would render the associated code unused. One simple method to generate an unique code for every tuple in the relation is to use a new integer as code for every newly created tuple. Therefore, an obvious way to implement this strategy would be to use positive integers as codes and keep track of the highest value of integer which has been used for coding a tuple in a particular relation. Then the code assigned to a newly created tuple would be the next higher integer. Let us call this next higher integer the 'next available code'.

The above-mentioned method for generating codes might lead to very long codes. Even for a relatively small relation on which update operations are very frequent, the generated codes for the newly inserted tuples would become very long. A better way of generating codes would be (i) to keep track of all the deallocated codes and re-use them and, (ii) as explained in the previous chapter, to generate a relative code for a tuple in a tuple scheme that does not have a set or a sequence as a member of one of its attribute domains and to generate an absolute code for a tuple in a tuple scheme that has at least one such attribute domain.

3.2.2. Operations on a Semilattice Database

As with any other kind of database, operations on the data of a semilattice database can be divided into two kinds - (i) Query operations, (ii) Update operations. Query operations request retrieval of some data from a database, whereas update operations insert or delete data or request modification of the existing contents of a database.

A typical query processing procedure in a semilattice database consists of three phases. In the first phase, codes for the user-supplied values are obtained from the database by the DBMS. In the second phase, the query is processed using mainly the coded values found in the previous phase; and finally in the third phase these codes are decoded and user is supplied with an answer to his query.

For example, in the chip database, to answer the query "what all pins are there in the rectangular enclosed region $X_1 \leq x \leq X_2$ and $Y_1 \leq y \leq Y_2$?, first, a selection followed by a projection on the relation POINT is made which would give the codes of all the points that lie in the specified region. Let us call this temporary relation T1. The next step would be to find the join of this relation with the relation PIN over the attribute *place*. The resulting relation (say T2) contains tuples each of which consists of the class of the pin, pin number and, a code for its place. Now to print the answer, *place* would have to be decoded from the relation POINT.

However, not all queries require all the three phases during their processing. Also a query could be processed in such a way that it might require a different combination or sequence of the above_mentioned three phases. For example the query "print the names of all the members that have balance <= $10", would not require the second and third phases.

Therefore, in order to function efficiently, a database management system should be equipped with data structures and access mechanisms that allow handling of all kinds of likely retrieval requests with a reasonable efficiency. The following subsections discuss the three very common types of retrieval requests and briefly mention others.

3.2.2.1. Exact Match Retrievals

The simplest type of query in a relational file is the exact match query - "is a specific tuple (defined by its key) in the relation"? A point worth mentioning here is that the semilattice data model does not rely upon the existence of keys in its relations because keys might break down. Therefore, a query posed by a user is not regarded as an exact match query unless exact values of all the attributes of the desired tuple are specified. However, the semilattice code of a tuple can be regarded as a key or the tuple identifier because a tuple can be uniquely identified by its semilattice code. During the processing of queries the code of a tuple may be used for retrieving data from relations. As an example, let us consider the query "what all pins lie in the rectangular region bounded by lines $x = X_1$, $x = X_2$, $y = Y_1$, $y = Y_2$"? As discussed earlier, during phase three of processing this query, the relation POINT is accessed to find actual values of co-ordinates corresponding to given codes for tuples in the POINT relation. Another instance of exact match query would be when values of all the attributes are specified by the user. For example, in the chip database the processing of the query "What is the class (input or output) of the pin located at the point $x = X_1$ and $y = Y_1$"?, would include an exact match retrieval on the relation POINT.

3.2.2.2. Partial Match Retrievals

A more complicated type of query in a multi-attribute relational file is a partial match query in which values of a subset of keys are specified. In general if a key has k attributes then in a partial match query the values of t of the k attributes ($t < k$) are specified and retrieval of all records that have those t values, independent of the other (k-t) values, is requested. Viewed in the context of a semilattice database system, a partial match query would specify values of a set of the attributes (coded or uncoded) that form only a proper subset of the key. For example in the HVFC database the query "print the names of all the members with a balance of $0.00" is a partial match query on the relation member. As another example the query "How many orders have been placed by members with a negative balance"? would require a partial match retrieval on the relation MEMBER as well as the relation ORDER.

3.2.2.3. Range Retrievals

In a range query we specify a range of values for several or all of the k attributes of the the relation, and all records that have every value in the proper range are then reported as the answer. For example one might be interested in querying a student database to find all students with grade point average between 3.0 and 3.5, age between 19 and 21 years, and parent's income between $15,000 and $25,000.

3.2.2.4. Other kinds of Retrievals

The three kinds of retrieval requests that have been investigated above are the most commonly discussed retrievals in database applications. However, other kinds of retrieval requests do occur. One such type is a 'best match retrieval' request. In some database applications one would like to query the database and find that it contains exactly what one is looking for; a builder might hope to find that he has in his warehouse exactly the kind of steel beams he needs for the current project. But often the database would not contain the exact item. The user then will have to settle for a similar item. The most similar item to the desired is usually called the "best match" or the "nearest neighbor" to the desired record. Here the "best" could be defined in terms of a distance function, which could be based on a general description of attributes. More formally, given a distance function D, a collection of points B (in a k-dimensional space), and a point P (in that space), it is often desired to find P's nearest neighbor in B. The nearest neighbor Q is such that

$$(\forall R \in B)\{(R \neq Q) \rightarrow [D(R,P) \geq D(Q,P)]\}$$

A similar query might ask for the K nearest neighbors of P (K>1).

3.2.3. References in a Semilattice Database System

In a semilattice database system, the value of an attribute in a tuple can refer to tuple(s) in a different relation. There are several implied assumptions for this kind of reference. The first assumption (the uniqueness assumption) is that the referenced tuple is unique. This is necessary in order to get a precise answer to the same query when it is posed more than once on the same instance of the database. The second assumption (the consistency assumption) is that the tuple being referenced should not change as a result of the old tuple getting deleted and a new tuple with the same code being inserted. This might happen if deleted codes are not handled properly. The third assumption (the existence assumption) is that the referenced tuple exists.

Any proposed data structure for the semilattice data model has to ensure that means for preserving the validity of these assumptions is provided.

3.2.4. Summary of Requirements Analysis

In short any proposed data structure for a semilattice database system should satisfy the following requirements:

(1) A dynamic mechanism should be provided for generating codes for newly inserted tuples which is capable of re-using the previously deleted codes.

(2) The proposed data structure should facilitate execution of various kinds of frequently desired operations on a database with reasonable efficiency. This includes query operations (exact match, partial match etc.) as well as update operations (insert, delete etc.).

(3) It should provide means to preserve and check the validity of all the above mentioned implied assumptions for references to other tuples.

3.3. Compressed-B^*-tree : The Proposed Data Structure

In this section a variant of the B-tree data structure is proposed for use in semilattice database system environments. Originally, B-trees were studied in the early 1970s by Bayer and McCreight [BaM72]. The B-tree structure showed early promise for efficient storage of very large indexes on secondary storage and allowed very fast random retrieval of records. New variants of B-tree structures and their applications continue to evolve. The general characteristics of a B-tree structure were discussed in chapter 2. A study by Comer [Com79] discusses basic B-trees and many of their variants.

A variant of the fixed-length-record B-tree structure suitable for variable length records was introduced by McCreight in 1977 [McC77]. Our Compressed-B-tree structure is based on this structure. It is named Compressed-B^*-tree structure because :(i) it uses a data compression scheme (to be described later), (ii) it maintains a minimum of 66% storage efficiency, a property of B^*-trees. The C-B^*-tree structure differs from the structure proposed by McCreight in the following ways:

(1) All records reside only at leaves. Node pages consist only of keys and pointers, i.e. a road-map to enable rapid location of records.

(2) All the pages at a certain level are ordered. In addition, all the pages at the leaf level are bidirectionally linked as shown in figure 3.1.

(3) The structure is designed to support retrieval on primary as well as secondary keys.

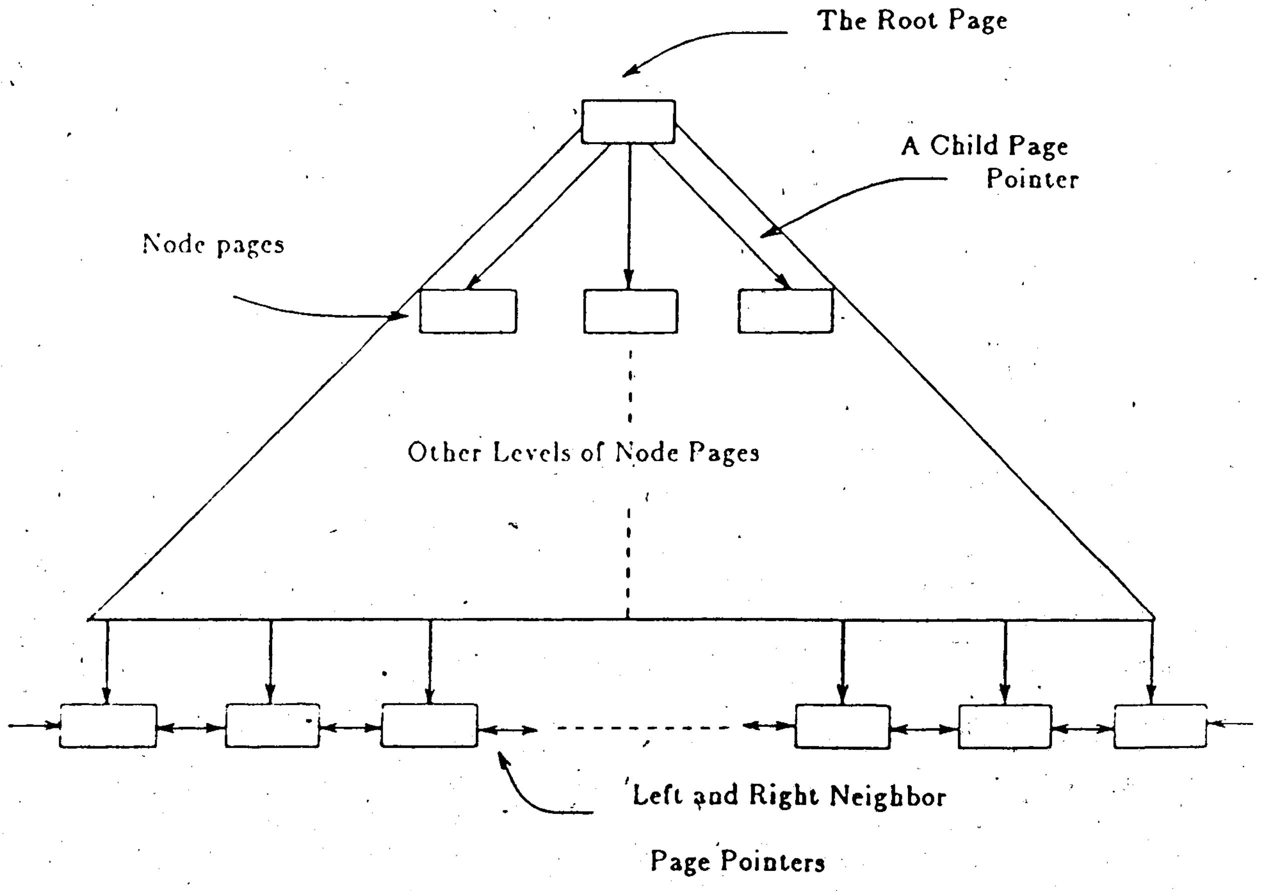
The Root Page
A Child Page Pointer
Node pages
Other Levels of Node Pages
Left and Right Neighbor Page Pointers

Figure 3.1 Structure of C-B^*-tree

(4) When used for retrieval on a secondary key, a compression scheme is employed which does not repeat a key common to many records that reside on the same page. There is no limit to the number of records that have the same key and

records having the same key value may be distributed over an arbitrary number of pages.

(5) The pagination strategy is modified ~~which~~ results in a significant performance improvement over the strategy proposed by McCreight[McC77].

3.3.1. Description of C-B^*-tree Structure

A C-B^*-tree structure is a generalization over B^+-tree and B^*-tree variants of B-tree. The records may have variable lengths. Also, the index could be on a primary key field or on a secondary key field. For the purpose of C-B^*-tree structure, a record consists of two parts : a key part which could be either a semilattice code, or some domain which can be ordered (e.g., integer numbers), and a data part which might be some property of the key, or might be a pointer to another record in the database etc. Since we are dealing with a structure which can support retrieval on primary as well as secondary key fields several records may have the same key value but no two records in the same C-B^*-tree are allowed to have the same key value as well as the same data field value. A C-B^*-tree with page length of p words (or bytes) has the following properties:

(a) If only child page pointers are followed, then, each path from the root node to a leaf node has the same length, h, the height of the C-B^*-tree (i.e. h is the number of nodes from the root node to the leaf node, both inclusive).

(b) In a node page the sum of lengths of all key parts and inter-page pointers cannot exceed p words, and (except for the root page) can be no less than p/2 words. Similarly, in a leaf page the sum of lengths of all key parts and data parts cannot exceed p words and can be (except for the root page) no less than p/2 words.

(c) For each non-root and non-leaf page P, the average number of words used per offspring page of P is at least 2p/3.

(d) The root may have at least one key part and two inter-page pointers or at least one complete record.

3.3.1.1. Structure of Interpage-pointer, Data Part and Key Part

The structures to represent an interpage-pointer, the key part of a record and the data part of a record in the C-B^*-tree are shown in figures 3.2(a), 3.2(b), and 3.2(c) respectively. All indicated lengths are in bytes. An inter-page pointer could be used to point to a neighboring page or to a child page.

3.3.1.2. Structure of a Non-leaf (or Node) Page

The general structure of a non-leaf page of a C-B^*-tree is shown in figure 3.3. P_0, P_1... etc. are child page pointers which point either to non-leaf or leaf pages. For brevity pointers to neighboring pages are not shown. K_0, K_1,...etc. represent values of key parts of records. m, n, r, t and, z are arbitrary positive integer numbers. A non-leaf page is structured according to the following rules:

(a) Keys within a page are sorted in ascending order of value, i.e. $K_0 < K_1 < \ldots K_r < K_{r+1} \cdots$.

(b) All the child page-pointers that occur before the key value K_r point to pages that lead to leaf pages with key values less than or equal to K_r.

(c) All the child page pointers situated between K_r and K_{r+1}, (except possibly the last child page pointer $P_{r+1,t}$) lead to pages containing only key values equal to K_r. The child page pointer $P_{r+1,t}$ leads to at least one page which has at least one record having a key value equal to K_r.

Length of the Following Field = L	Pointer Value
1 Byte	L Bytes

Figure 3.2(a): Structure of an Interpage Pointer

Length of the Following Field = l	Length of the Following Field = L	Key Value
1 Byte	l Bytes	L Bytes

Figure 3.2(b): Structure of a Key Part

Length of the Following Field = l	Length of the Following Field = L	Data value(s)
1 Byte	l Bytes	L Bytes

Figure 3.2(c): Structure of a Data Part

Figure 3.2 Structure of an Interpage Pointer, Key Part and Data part

3.3.1.3. The Structure of a Leaf Page

The general structure of a leaf page of a C-B^*-tree is shown in figure 3.4. Again K_1, K_2, ... etc. represent the values of the key parts of records whereas $D_{j,k}$s (j and k are positive integers) represent data parts. The following rules govern the structure of a leaf page:

(a) Keys within the same page are organized in ascending order of value, i.e., $K_0 < K_1 < \cdots < K_r < K_{r+1}$.

(b) A complete record can be formed by combining a data part with the key part that immediately precedes it.

An example of C-B^*-tree is shown in figure 3.5. A star (*) represents a child page pointer whereas a # represents a data part. Key values are represented by integers.

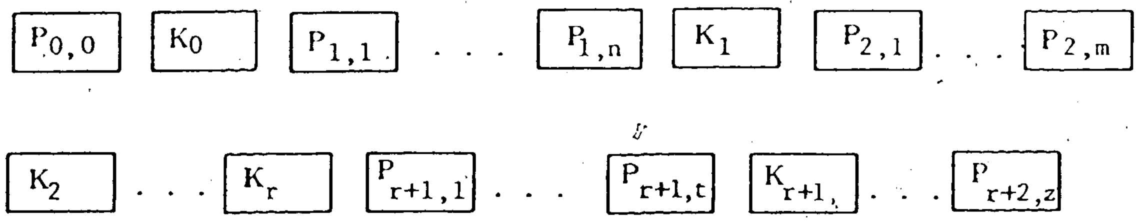

Figure 3.3 .Structure of a Non-leaf Page

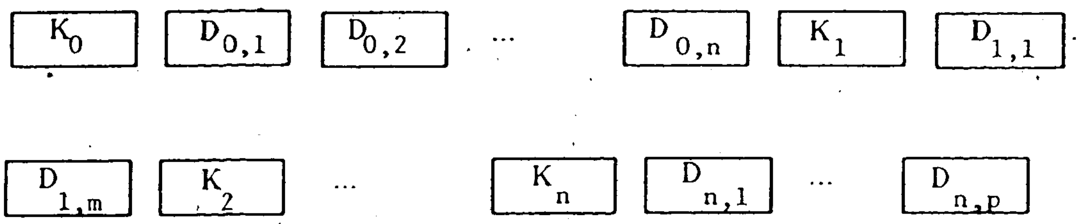

Figure 3.4 Structure of a Leaf Page

3.4. Operations on a C-B^*-tree

Because of variable length records and use of the compression scheme outlined above, the algorithms for various operations like search, insert, delete etc. in a C-B^*-tree are somewhat different from those in a B-tree with fixed length records. Also, during insertion or deletion of a record, a phenomenon called anomalous overflow or underflow might take place. This phenomenon cannot occur in a B-tree with fixed length records. This phenomenon is explained later. Here, first of all, some terminology for the C-B^*-tree is defined.

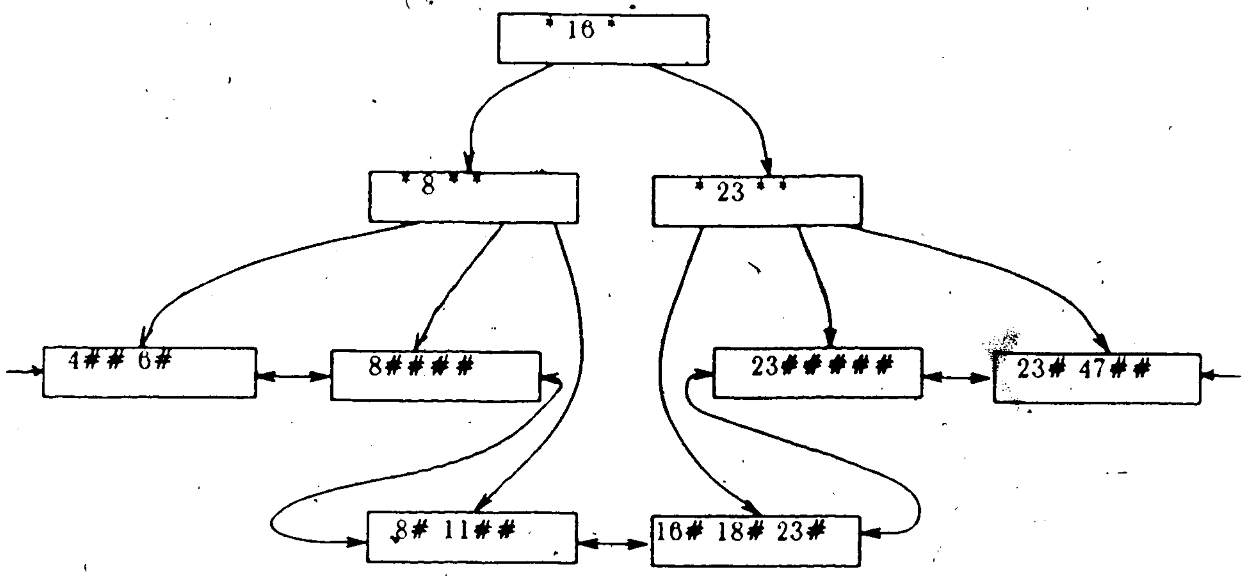

Figure 3.5 An Example of C-B^*-tree

A Scroll is exactly like either a nonleaf or leaf page of a C-B^*-tree page but it is not constrained in length, while a C-B^*-tree page must contain between p/2 and p words (p is the size of the page). A **boundary point** is either the beginning of a key part or beginning of a data part (beginning point of child page pointer in case of a node page) that is not immediately preceded by a key part. A sequence of boundary points on the scroll determine a **feasible pagination** of the scroll if each sequence of scroll records that lie properly between the beginning position of the scroll and the first boundary point, the last boundary point and the end position of the scroll, and between each pair of consecutive boundary points all form valid C-B^*-tree pages, i.e., all contain between p/2 and p words. A **pagination strategy** is an algorithm for choosing from a scroll a particular sequence of boundary points that determine a feasible pagination of the scroll.

A noteworthy point here is that when a scroll is paginated into several pages, the combined occupied length of all the pages into which the scroll is split could be greater than the original length of the scroll. This is due to the fact that for every data part (or child page pointer), the associated key part must reside on the same page. Therefore, if a data part (or a child page pointer) is selected as a boundary point, the key associated with this data part (child page pointer) is inserted before splitting the scroll. This point will become clearer from the example shown below. When the scroll of figure 3.6(a) is split into the three pages as shown in figure 3.6(b), the common keys 8 and 11 are supplied at the beginning of the second and third page respectively.

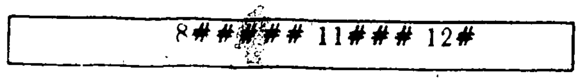

Figure 3.6(a)

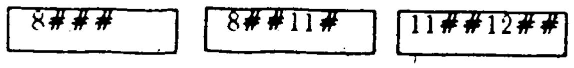

Figure 3.6(b)

Figure 3.6 Composition and Pagination of a Scroll

Similarly, when certain pages are combined to form a scroll, the length of the resulting scroll could be less than the combined occupied length of all the pages. For example, when the three pages shown in figure 3.6(b) are combined into the scroll as shown in figure 3.6(a), the length of the scroll would be less than the combined occupied length of the pages.

Now we are in a position to examine the algorithms for various operations on C-B^*-tree.

3.4.1. Search Operation

Random search for the desired records always begins at the root page. If the page that is currently being searched is a node page, it is searched sequentially until either the end of the page is reached or, a key part is encountered whose key value is greater than the key value of the desired records. If the end of page is reached then the last child page pointer is followed. Otherwise, if a key part is encountered whose key value is greater than the key value of the desired record(s), then the child page pointer immediately preceding this key part is followed.

If the page being searched is a leaf page, the page is sequentially searched until a key-part with a key value higher than the given key value is encountered. If end of page is reached before such a key part is encountered then there is no record with the given key value. Otherwise, if the value of the key part immediately preceding the current key part is not the same as the given key value then there is no record with the given key value. If the value of the key part immediately preceding the current key part is the same as the given key value, the record is returned for checking the values of fields in the data part.

The random search algorithm works in such a way that if there are several records having the given key value then the record that is situated rightmost in the tree is picked first. Now, since the pages of the C-B^*-tree that are at the same level are linked both ways, the search for the rest of the records having the given key value proceeds leftwards and all records are returned for further examination, until a record is found with a different key value.

For example, in the C-B^*-tree of figure 3.5, let us suppose the records with the key value of 23 are desired. In the root page end of page is reached before we arrive at the key value higher than 23. Therefore, the last child pointer is followed. At the second level, again, the end of page is reached before a key value higher than 23 is encountered. Again the last child page pointer is followed. Now, at the next level, which is the leaf level, the page is searched until the key value 47 is encountered. Therefore the record previous to this key part is examined. This is a record with key value 23. The record is returned. Now, since this record is the first record in the current page, the page to the left of the current page is examined for more records with the given key value. In this page five more records with the given key value are found. Since again the first record in the current page has the given key value, the page situated to the left of it is examined for more records, where one more record with the given key value is found. After that, a record with the key value of 18 is encountered. Further search for more records having the given key value then stops.

3.4.2. Insert Operation

To insert a record into a C-B^*-tree, first a search is made for it. If the record already exists, an error condition is returned. Otherwise, the record is inserted in the current page. Insertion of a record might cause a page P to overflow. In that case a scroll consisting of at most two neighboring sibling pages and the page P is made. The order of preference in the selection of neighboring pages is shown in figure 3.7. In some cases there may not be a third sibling page of P. In that case a scroll consisting of only two sibling pages is made. An attempt is made to paginate this scroll without increasing the total number of pages in the tree. If it succeeds, the operation of insertion is complete. Otherwise, a new scroll consisting of either pages P and P_2, or, pages P and P_1 (and also the intervening key parts in case P is a node page) is made. This scroll is then split into three pages. This split might cause an overflow (or even underflow in some cases) which is handled recursively.

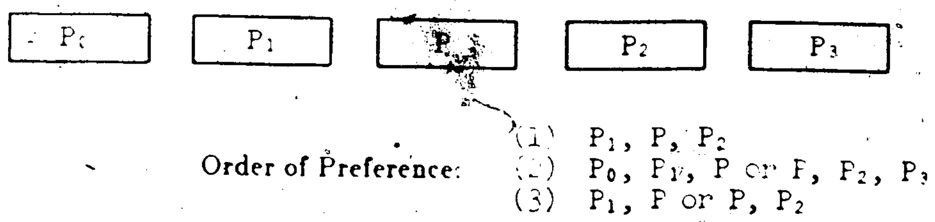

Figure 3.7 Selection of Neighboring Pages for Making a Scroll

3.4.3. Delete Operation

The operation of deleting a record from the C-B^*-tree is handled in a way that is quite similar to the insert operation. If the desired record is not found, an error condition is returned. If the record is found, it is deleted. If this deletion does not cause the current page, P, to underflow, the operation of deletion is complete. Otherwise, a scroll consisting of at most two neighboring sibling pages (and intervening key part from the parent page, if page P is an internal node), as described in case of insert operation, is made. Now, an attempt is made to paginate this scroll so as to decrease the total number of pages in the tree by one. This might cause a parent page to underflow (or even overflow in some cases), a situation that is handled recursively. Otherwise, the scroll is paginated into three pages.

3.4.4. Anomalous Overflows and Underflows

In a C-B^*-tree, a parent page P sometimes may overflow (underflow) when a record is deleted from (inserted into) one of its child pages or, a scroll consisting of two or three children pages of P (and intervening key parts in case children pages are node pages) is repaginated. This is called anomalous overflow (underflow). This phenomenon is also possible in B-trees with variable length records. However, it has not been explained earlier.

Let us consider the tree shown in figure 3.8(a). A record with key value K is inserted at the end of page D which makes it overflow. The tree after repagination of records in pages B, C, and D is shown in figure 3.8(b). Now if the length of key K is less than the length of K_3, then it is possible that page A could underflow. This might happen if the length of $(K_1 + K_2 + K)$ is less than p/2. This is an example of an anomalous underflow. Similarly, deletion of a record from a page P and subsequent pagination of a scroll consisting of P and one or two neighboring pages of P may cause the parent page of P to overflow.

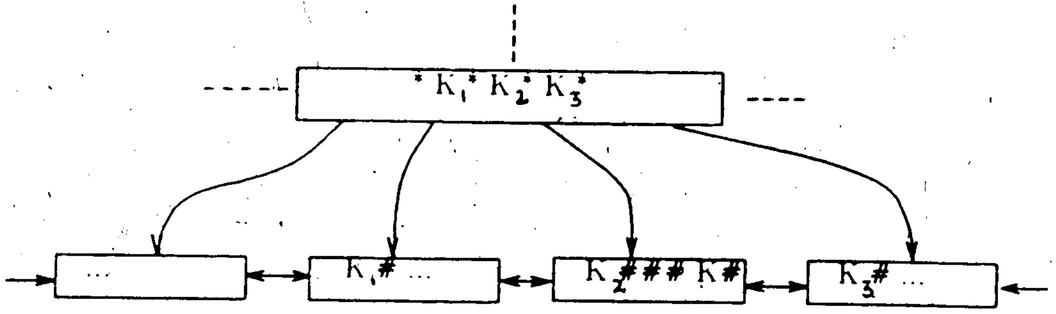

Figure 3.8(a)

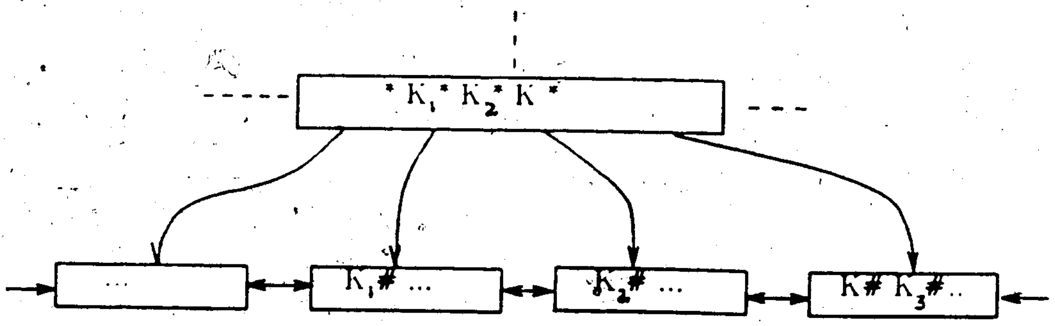

Figure 3.8(b)

Figure 3.8 Part of a C-B^*-tree to Illustrate Anomalous Underflow

3.5. Organization of a SLDB Relation into C-B^*-tree Files

In order to meet various requirements outlined in section 3.3, the data and other associated information of a SLDB relation is organized into three different forms of C-B^*-tree files. Each of these forms is designed to meet a specific part of the requirements. These forms are : (i) encoding C-B^*-tree file, (ii) decoding C-B^*-tree file and (iii) secondary index C-B^*-tree file. The structure and purpose of each of these files is explained in the following subsections. It should be noted however, that only the decoding file is required to represent all the information in a relation. The other two kinds of files are there for greater efficiency.

3.5.1. The Encoding C-B^*-tree File

A different encoding file is associated with each of the relations of a SLDB system. The purpose of the encoding file is to facilitate efficient generation of a code for a newly inserted tuple of the relation and re-use of the deallocated codes. From an encoding point of view the relations in a SLDB system can be classified into two types: relations whose tuples are coded using the absolute coding scheme and relations whose tuples are coded using the relative coding scheme. These coding schemes and reasons for selecting a particular coding scheme for coding tuples in a given relation were explained in Chapter 2.

Let us first consider the case where an absolute code is assigned to each tuple of a relation. The encoding file for such a relation has codes or tuple identifiers as keys. There is no data part. All the deallocated codes plus a special integer number called 'next available code' form the records of the encoding file.

Initially, when there are no tuples in the relation, the tree has only one key with value equal to zero. This key indicates the 'next available code'. As records are inserted, this key is incremented. Now if a tuple which had the highest code value in the relation (i.e. less by one than the 'next available code') is deleted from the relation then the value of the 'next available code' is decremented by one. If a different tuple is deleted, the deallocated code is simply inserted in the proper order into the encoding file. Later, when a new tuple is inserted into the relation, the smallest key value is deleted from the encoding file and is used as code or tuple identifier for the newly inserted tuple. This kind of garbage collection on codes permits the use of shorter codes than would be otherwise possible in a dynamic database.

For example, let us consider a relation which has some non-atomic domain(s) in its tuple-scheme (e.g. the SUPPLIER relation described earlier in this chapter). The evolution of the encoding C-B^*-tree file for this relation is shown in figure 3.9. Initially, the file contains one key with value zero (fig. 3.9(a)). Figure 3.11(b) shows the contents of the file after continuous insertion of ten records. Let us suppose records with codes 0,3,5,7, and 8 are deleted from the relation. The resulting structure of the encoding C-B^*-tree file is shown in figure 3.9(c), assuming that a tree page can hold at most two keys and three interpage pointers. Now the tuple with the code 9 is deleted. Since 9 is the highest code used, the next available code is decremented. This is shown in figure 3.9(d). Now if a new record is inserted, the smallest code 0 is deleted from the encoding file and used as the code for the newly inserted tuple (fig. 3.9(e)).

Handling of the process of code generation and re-use of deallocated codes is slightly more complicated in the case of relations whose tuples are coded using the relative coding scheme. This is because of the fact that the generated code is relative to the value of all the referenced attributes. For example let us consider the instance of ORDER relation of the HVFC database given in Appendix A. A single code is generated for *order-no* and *quantity* relative to the codes for *purchaser* and *item*.

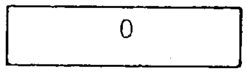

Figure 3.9(a)

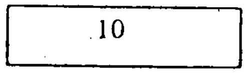

Figure 3.9(b)

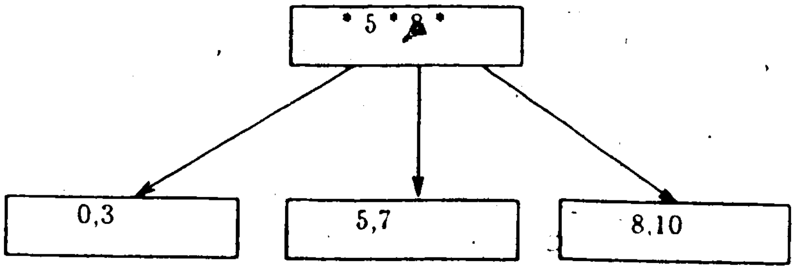

Figure 3.9(c)

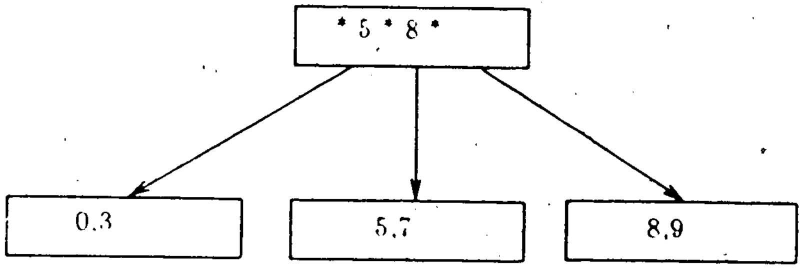

Figure 3.9(d)

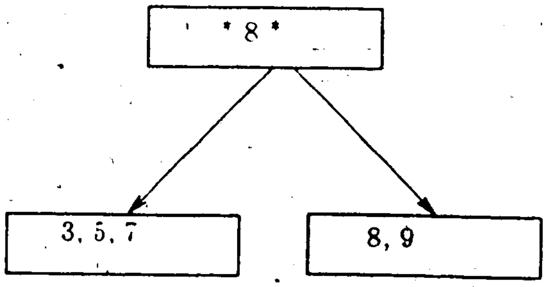

Figure 3.9(e)

Figure 3.9 Evolution of an Encoding File

In such cases the combined codes for the referenced data (in a certain order) constitute the key for a record in the encoding file whereas each of the associated deallocated codes forms the data. For example, in the encoding file for the ORDER relation of HVFC database, concatenated codes for *purchaser* and *item* are the keys, and the data are codes for *order-no* and *quantity*. When a new tuple is inserted into the relation the combined code for its referenced parts is used as the key to search the encoding C-B^*-tree. If the search fails then a record with the given key and the integer 1 as data is inserted into the encoding file and code 0 is used as relative code for the tuple. If there is only one record having the given key value, the value of its data part is incremented by one. If there is more than one record having the given key, the record with the lowest value of data part is deleted and this value is used as relative code for the inserted tuple. When a tuple is deleted, the referenced code of the deleted tuple is used as key to search the associated encoding file. If the relative code for this tuple is one less than the next available code for the given referenced code, the value of the next available code is decremented by one. If the value of the next available code is 1, the record is deleted from the file. otherwise, the record with the given key value and its associated relative code as data are inserted into the encoding file.

3.5.2. The Decoding C-B^*-tree File

Each relation in a SLDB system has one decoding C-B^*-tree file associated with it. The decoding files in a SLDB system are perhaps the most frequently searched files during various query and update procedures. The keys for a decoding file are made up of the complete codes or tuple-identifiers of the tuples in the relation. In case a tuple-identifier is composed of a combination of codes, the order in which the codes of various attributes are concatenated is specified in the knowledge base (or data dictionary) of the database system. For example, in the SUPPLIER relation of the HVFC database a single code serves as a tuple identifier whereas in the ORDER relation of the same database the codes for *purchaser* and *item* together with a single generated relative code for *order-no* and *quantity* in some specified order can be used as the tuple-identifier.

The data part for a decoding tree is made up of the values of the attributes in the uncoded part of the tuple together with some optional information e.g. the codes for the tuples in other relation(s) where this tuple is referred to. The possibility of storing various kinds of optional information within the data part of a tuple will be explored, in detail, in the next chapter.

3.5.3. The Secondary Index C-B^*-tree File

A relation in a SLDB system can have an optional number of secondary index C-B^*-tree files with up to a maximum of one secondary index file for each attribute. A coded or uncoded attribute of the relation may be used as the key for a secondary index file. The data part consists of a set of tuple identifiers or complete codes of all the tuples having the given secondary key value.

3.5.4. Using the C-B^*-tree Files

Encoding files are used for the process of allocating codes for the newly inserted tuples and re-use of deallocated codes as explained earlier. The process of answering a query depends very much on the type or composition of the query. Decoding files can be used to answer exact match queries that specify complete codes or tuple identifiers. The process of answering an exact match query that specifies the value of all the attributes, a partial match query or a range query is handled in a similar manner. If a query specifies a value (or a range of values) for one or more attributes of a relation for which secondary index files exist, the lists of codes corresponding to the values of these attributes are fetched. Next, these lists of codes are processed (by taking union ,intersection etc. depending upon the nature of the query) in order to get the list of codes for the target tuples. These tuples are then accessed using the decoding file associated with the relation. In case the query does not specify the value of any of the attributes of the relation for which a secondary file exists, target tuples are found by sequentially searching the leaf level of the decoding file.

The proposed organization of a SLDB relation can be effectively used for checking and maintaining the validity of three previously mentioned assumptions regarding references to tuples in other relations, namely: the uniqueness assumption, the cosistency assumption, and the existence assumption. The very method of generating codes for the newly inserted tuples ensures that a referenced tuple is unique. The second assumption that the referenced tuple should not change as a result of an old tuple being deleted and a new tuple being inserted in its place with the same codes, is very much tied up with the semantics of update operations in a SLDB system. Therefore the solution to the problem of checking the validity of this assumption rests in the hands of the logical part of the database management system. However, it is very likely that when a tuple in a SLDB relation is deleted or modified, the tuples that refer to the deleted/modified tuple would have to be accessed (for deletion, modification etc.). Accessing all tuples that refer to a given tuple can be made efficient by storing a list of codes of all such tuples as optional information along with the data part of the tuple. The validity of the third assumption (i.e. the existence assumption) can be maintained at the cost of a few more accesses at the time of each insert, delete and update operation.

3.6. Experimentation with the Implemented Structure

The *C-B*-tree* structure has been implemented under 4.2 BSD Unix operating system at University of Alberta. The host language is C. It has been implemented using the file structure and input-output system provided by the Unix operating system. An actual physical block of the C-B^*-tree also includes some overhead information such as the type of the page (node or leaf), the number of bytes of data present in the page, sibling page pointers etc. In order to determine a suitable paging strategy for the C-B^*-tree several experiments were performed. Their results are also applicable to other variants of B-tree with variable length records. These experiments and their results are explained in the following subsections.

As explained earlier, a **pagination strategy** is an algorithm for choosing from a scroll a particular sequence of boundary points that determine a feasible pagination of the scroll. A straightforward strategy is to divide the scroll into pages of as equal size as possible without splitting a record. This strategy is normally used with B-trees having fixed-length records. It is called ES strategy. In case of variable-length B-trees it means minimizing the difference between the size of the largest page of the pagination and the size of its smallest page.

Another pagination strategy, called MS, is based upon the general principle that shallowest trees result from having the shortest keys nearer to the root and longer keys nearer to the leaf level. To illustrate this principle let us consider the following ordered sequence of 25 keys : bbb, c, ee, f, gg, h ... w, xx, y, zz, 9. A possible arrangement of these keys into C-B^*-tree is shown in figure 3.10(a). It is assumed that each page has space for six characters. Further, for the sake of simplicity, it is assumed that the interpage pointers do not occupy space. Another possible organization for the same data is shown in figure 3.10(b). In the latter organization shorter keys have been used in node pages. This has resulted in a shallower tree which in turn implies fewer accesses per search (on average).

Formally, the MS strategy can be defined as a pagination strategy which chooses a pagination that minimizes the sum of the lengths of the boundary keys that are returned to the parent page. This definition of the MS strategy is correct for a variable-length B^*-tree that does not use any kind of compression scheme. Since in the C-B^*-tree the data compression also affects the size of the parent page the definition of MS strategy should be modified. In this case the MS strategy would choose a pagination that simply minimizes the size of the resulting parent page.

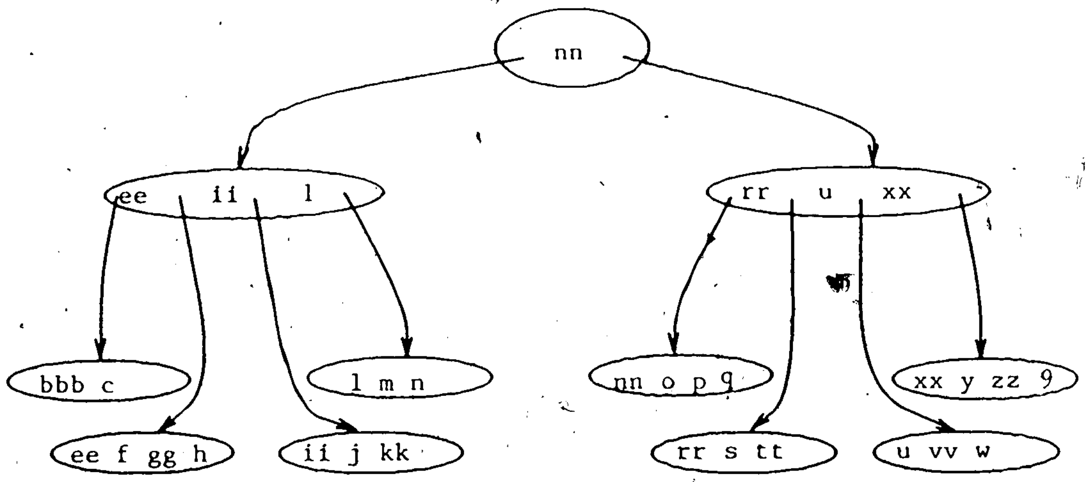

Figure 3.10(a)

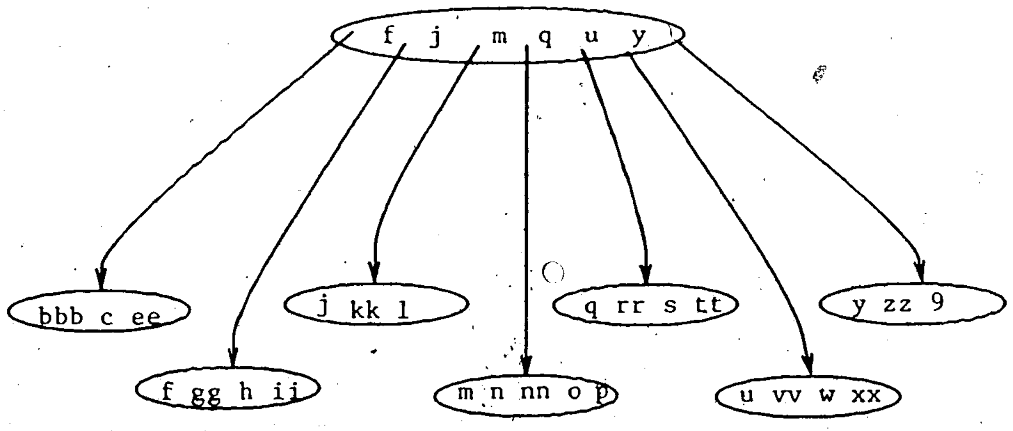

Figure 3.10(b)

Figure 3.10 Effect of Keeping Smaller Keys Nearer to the Root Level

Results of McCreight's experiments to compare the performance of the ES and MS strategies are shown in table 3.1 and table 3.2. In this experiment 33,381 records with an average size of 12.0 bytes were inserted into a B^*-tree using the MS strategy. The same insertion sequence was performed in another B^*-tree using the ES strategy.

Space Effic.	Level	Avg. Rec. Length	# of Pages	Avg. Recs./Page
1120 pages or space efficiency of 82%	1(Root)	12 chars.	1	1
	2	12.8	2	17
	3	12.5	36	29.0
	4(leaf)	12.0	1081	29.9

Table 3.1 Performance of the ES Strategy

Space Effic.	Level	Avg. Rec. Length	# of Pages	Avg. Recs./Page
1141 pages or space efficiency of 81%	1(Root)	7.0 chars.	1	28
	2	9.5	29	37.3
	3(leaf)	12.1	1111	29.0

Table 3.2 Performance of the MS Strategy

The statistics shown in table 3.1 and table 3.2 support the fact that the MS strategy helps to produce shallower trees as compared to the ES strategy. Another observation is the slightly higher space efficiency resulting from the ES strategy. Perhaps it is difficult to associate any statistical significance with one observation of such a small difference. However, if the ES strategy is consistently found more space efficient than the MS strategy, it would be a significant conclusion since it would mean more space for records at the leaf level and, more significantly, more space for keys and pointers at node levels and consequently shallower trees. Experiments were done to verify this point. Two C-B^*-trees were built using the MS and ES strategies respectively. A fixed number of records in the same sequence were inserted into each tree. The same experiment was repeated for different block sizes. The resulting statistics are shown in table 3.3. These results evince that the ES strategy is in general more space efficient than the MS strategy.

3.6.1. The CS Pagination Strategy

In this section a modified version of the MS strategy called CS strategy (for Combined Strategy) is presented. The CS strategy exploits the higher space efficiency feature of the ES strategy. According to the MS strategy the pagination that results in the smallest page size of the parent page is selected. However, it does not offer any further choice between two candidate paginations that yield the same parent page size. The CS strategy on the other hand looks into the size of the pages produced by two paginations that yield the same size of the parent page and the pagination which is closer to having equal page sizes is preferred. For instance, let us suppose P_b is the best among so far examined paginations of a given scroll and P_t is the pagination currently under examination. If the pagination defined by P_t results in a parent page whose size is less than the parent page size given by pagination P_b, the pagination P_t replaces P_b as the best pagination. If P_b and P_t result in the same size of the parent page, the pagination which has a smaller difference between its largest page and smallest page will be preferred. Since, in addition to promoting smaller keys nearer to the root level, the CS strategy also attempts to divide the scroll into pages of as equal size as possible, it can be hoped that it would have higher space efficiency than the MS strategy and would also result in shallower trees than both ES and CS strategies.

3.6.1.1. Performance Evaluation

Several experiments were performed in order to evaluate the performance of the ES, MS, and CS strategies. In all these experiments randomly generated different integer numbers were used as keys. The key lengths were varied by padding these integer numbers on the left by a random number of zeros. In this manner keys whose lengths varied from 7 to 21 were generated. The node pages had interpage pointers that were five bytes long. To simplify analysis same amount of space was left for dummy data parts in leaf pages. Also, the same amount of space (18 bytes) was left in both node and leaf pages for overhead information. In each of the experiments three C-B^*-trees were built using the ES, MS, and CS strategy respectively. For a particular experiment the same insertion sequence was used for building up each of the three trees.

The first set of experiments was performed in order to verify the assumption that the ES strategy is slightly more space efficient than the MS strategy. The results of these experiments are shown in table 3.3.

Block	ES Strategy			MS strategy		
Size	Node	Leaf	Overall	Node	Leaf	Overall
320	80.4	82.9	82.7	79.7	82.2	82.0
384	80.2	82.2	82.1	77.0	80.6	80.4
448	79.2	84.9	84.6	78.9	83.4	83.2
512	81.3	85.2	85.0	80.9	83.0	82.9

Table 3.3 Comparision of Space Efficiency of the Two Strategies

In the second set of experiments the growth of three C-B^*-trees using the ES, MS and CS strategies respectively was analyzed in detail. A block size of 512 bytes was chosen for each of the trees. These statistics are shown in table 3.4 through table 3.9. The first snapshot of the three trees was taken after the of insertion of 12,811 keys. By this time the number of levels in the ES tree had grown to four whereas number of levels in the MS and CS trees was three. The next snapshot was taken after the insertion of 21,853 keys. At this time both the MS and ES trees had four levels whereas the CS tree had only three levels.

Space Effic.	Level	Avg. Rec. Length	# of Pages	Avg. Recs./Page
625 pages or space efficiency of 85.04%	1(Root)	20 chars.	1	1
	2	14.96	2	12.50
	3	14.97	27	21.03
	4(leaf)	14.50	595	21.53

Table 3.4 Statistics of ES Tree After Insertion of 12,811 Records

Space Effic.	Level	Avg. Rec. Length	# of Pages	Avg. Recs./Page
634 pages or space efficiency of 82.93%	1(Root)	7.80 chars.	1	21
	2	9.91	22	26.77
	3(leaf)	14.5	611	20.96

Table 3.5 Statistics of MS Tree After Insertion of 12,811 Records

Space Effic.	Level	Avg. Rec. Length	# of Pages	Avg. Recs./Page
631 pages or space efficiency of 83.38%	1(Root)	8.76 chars.	1	21.00
	2	10.29	22	26.63
	3(leaf)	14.50	608	21.07

Table 3.6 Statistics of CS Tree After Insertion of 12,811 Records

Space Effic.	Level	Avg. Rec. Length	# of Pages	Avg. Recs./ Page
1076 pages or space efficiency of 84.19%	1(Root)	17.0 chars.	1	2.0
	2	16.91	3	15.0
	3	14.94	48	20.33
	4(leaf)	14.48	1024	21.34

Table 3.7 Statistics of ES Tree After Insertion of 21,853 Records

Space Effic.	Level	Avg. Rec. Length	# of Pages	Avg. Recs./ Page
1084 pages or space efficiency of 82.68%	1(Root)	8.0 chars.	1	1.0
	2	8.91	2	17.5
	3	10.14	37	27.21
	4(leaf)	14.48	1044	20.93

Table 3.8 Statistics of MS Tree After Insertion of 21,853 Records

Space Effic.	Level	Avg. Rec. Length	# of Pages	Avg. Recs./ Page
1059 pages or space efficiency of 84.57%	1(Root)	8.11 chars.	1	34.0
	2	10.24	35	28.22
	3(leaf)	14.48	1023	21.36

Table 3.9 Statistics of CS Tree After Insertion of 21,853 Records

The access cost of a tree (defined as the average number of page accesses required in order to search a key) depends upon the number of levels in the tree. Let us suppose two page buffers are used during searching (one permanently dedicated to the root page and the other available for reading other pages). If the tree building process were stopped at 10,000 keys all three trees would have an equal access cost of 2. On the other hand at the end of insertion of 21,853 keys the MS and ES trees have an access cost of 3 whereas the access cost for the CS tree is still 2. This means an improvement of 50% in case of the CS tree. However, the extent of improvement will also depend upon the number of page buffers available in memory. Let us assume 16 page buffers are used and these page buffers are allocated to those pages whose probability of access is the highest. This means that in case of the ES tree the root page, all the 3 pages at the second level and 11 pages at the third level will be assigned permanent buffers. Therefore the number of page-in operations per search would be approximately $(\frac{37}{48})\times 2+(\frac{11}{48})\times 1=1.77$. In case of the MS tree root plus both the pages at the second level and 12 of the pages at the third level will be assigned permanent buffers, so average number of page-in operations per search would be $(\frac{25}{37})\times 2+(\frac{12}{37})\times 1=1.67$ This means an improvement of about 5.65% over the ES tree. On the other hand in ca[illegible] the CS tree root plus 14 of the pages at the second level will be assigned permanent buffers. The average number of page-in operations per search would therefore be $(\frac{21}{35})\times 2+(\frac{14}{35})\times 1=1.6$. This means an improvement of 4.2% over the MS tree and an improvement of 9.6% over the ES tree. This is quite in contrast with the earlierly calculated improvement of 50%.

We define **data capacity** as another measure of the performance of a pagination strategy. For a given height of the tree the data capacity of a tree can be measured as the maximum number of words that the tree can hold without requiring an increase in its height. Like the access cost measure the data capacity measure also depends upon the pagination strategy used for generating the tree, the nature of data, its insertion sequence and the block-size used. However, the data capacity measure gives a better performance measure of a strategy since it does not depend upon the point where the tree building process is stopped. Also, the data capacity of a tree does not depend upon the amount of other resources (such as page buffers in case of access cost) that are available for searching the tree. However, the two measures are not unrelated as the data capacity in fact measures the ability of a pagination strategy to generate shallow trees, and a shallow tree in turn means smaller access cost.

The data capacity for three C-B^*-trees generated using the ES, MS and CS strategies respectively was estimated for the same kind of data as described in section 6.1.1. The same insertion sequence was used in generating each of the trees. The results of this experiment are shown in table 3.10 where data capacity has been exprssed in terms of number of records and the average record length is 14.5 bytes.

Block-size	ES Strategy	MS Strategy	CS Strategy	% impv. with CS
320	2789	4743	4953	4.3
384	4406	8548	9151	7.0
448	7297	14727	16093	9.2
512	12766	21779	23452	7.7

Table 3.10 Comparision of Data Capacity of the Three Strategies

Block-size	Node Level	Leaf Level	Overall Level
320	80.4	82.9	82.7
384	80.2	82.2	82.1
448	79.2	84.9	84.6
512	81.3	85.2	85.0

Table 3.11 Space Efficiency of the ES Strategy

Block-size	Node Level	Leaf Level	Overall Level
320	79.7	82.2	82.0
384	77.0	80.6	80.4
448	78.9	83.4	83.2
512	80.9	83.0	82.9

Table 3.12 Space Efficiency of the MS Strategy

Block-size	Node Level	Leaf Level	Overall Level
320	79.0	82.0	82.0
384	81.7	80.8	80.8
448	87.3	82.8	82.8
512	82.6	83.4	83.4

Table 3.13 Space Efficiency of the CS Strategy

3.7. Storage Efficiency and Pagination Strategy

This last experiment is concerned with comparing the three strategies for their storage efficiencies. For each block size, three C-B^*-trees were generated using the ES, MS, and CS strategies. In each case the same 15,000 keys were inserted in the same sequence. The results of this experiment are given in table 3.11 through table 3.13. On the basis of these results we can draw the following general conclusions:

(1) For the internal node pages the CS strategy seems to have the highest storage efficiency followed by the ES strategy. This is possible because in addition to promoting shorter records toward levels that are nearer to the root page, the CS strategy also tends to divide pages into equal sizes. These smaller records tend to get packed much more compactly than longer records as in the case of the ES strategy. Also, since these pages are more likely to be equal than in case of the MS strategy, the internal node level of the CS tree benefits from the slightly better storage efficiency of the ES strategy. This small difference consequently results in a significant improvement in the data capacity of a tree generated using the CS strategy.

(2) As far as the overall storage efficiency and the storage efficiency at the leaf level are concerned, the ES strategy appears more space efficient than MS and CS strategies. There is almost no difference between storage efficiency of the CS and MS strategies. Logically, the CS strategy should have slightly better overall storage

efficiency since it has the highest storage efficiency at the internal node level. However, it appears that since in a C-B^*-tree a small fraction of the total data is stored at the internal node level, a slightly higher storage efficiency at the internal node level is not able to contribute much towards the overall storage efficiency. Also, the two strategies are so close in their storage efficiencies that even the allocation of a couple of pages for C-B^*-tree files of as small sizes as ours has significant effect upon the overall storage efficiency or upon the storage efficiency at the leaf level.

3.8. Conclusions

A single storage structure and its associated access mechanisms presented in this chapter can support all kinds of operations on a semilattice database system. Generally, database management systems offer more than one option for the organization of data to suit various kinds of applications. However, since this is only the Phase Zero of the development of a semilattice database management system, it would be worthwhile to wait for the experience gained by running this prototype DBMS in a real environment before we embark upon any kind of extension or modification of the existing physical data organization facilities. Recent experience with the development of some other experimental database management systems indicates that experience gained during the use of the initial implementation in a real environment provides invaluable insights into the requirements of the DBMS as well as the shortcomings of the running system [Cha81, SEP77, Sto80].

The CS pagination strategy presented in this chapter certainly has an edge over the MS strategy. The results are based upon experiments done with the C-B^*-tree. However, it is not difficult to realize that these results also apply to other variants of variable-length record B-tree structures e.g. the B^+-tree, prefix B-tree etc. The extent of improvement is likely to depend upon the blocking factor as well as the range of possible key lengths. The determination of the exact nature of this dependence is left for future work. However, our speculation is that the improvement in performance by using the CS strategy increases with increasing blocking factor or decreasing range of possible key lengths until a saturation state is attained after which further increase in blocking factor or decrease in the range of possible key lengths does not yield any further improvement.

Chapter 4

An Adaptive Data Structuring System: Feasibility Analysis

4.1. Introduction

One of the most difficult tasks faced by a database administrator is to select the access methods for a database system. Most of the time his/her decisions are based on intuitions and qualitative impressions. The design aids available for this task assume prior knowledge of most or all the applications for which the database system would be used along with their relative frequencies of use. In practice this is rarely the case. In most of the real world database environments that serve the interests of a variety of users, the usage of the database cannot be fully determined at the system design stage. Also, in some cases the pattern of usage may change as interests of users develop. Sometimes it might even necessitate restructuring of the complete database system.

This chapter presents the design of an adaptive data structuring scheme (ADSS) to alleviate these problems. The C-B^*-tree structure presented in the last chapter was designed keeping this goal in mind. The work presented in this chapter is preliminary in nature and therefore limited to giving an overview of the proposed design and doing a feasibility analysis. No algorithms for performing the specific steps are given. However, wherever necessary, guidelines are given and related works by other researchers are cited.

4.2. Adaptive Data Structuring System

The adaptive data structuring system is a self organizing data structuring system in which all the structuring of data is is not specified or performed at the initial data load time, rather data gets organized in a more and more efficient manner as the usage patterns on data become clearer. ADSS is capable of handling certain exceptional conditions in the data, e.g. the breakdown of a functional dependency. A further feature of the system is that incremental reorganization of data is possible. Of course, any change in the organization of data does not affect the conceptual scheme of the database system.

The principal components of ADSS in conjunction with other components of our proposed prototype semilattice database management system are shown in figure 4.1. The logical part of the database management system consists of a request interpreter and a semantics checker. All accesses to the semilattice database are through the request interpreter. This includes accesses by users to data stored in the database as well as various kinds of accesses by the database administrator. The request interpreter supports an interactive query language as well as running of pre-compiled application programs Various constraints on the data in a database system are checked and enforced by the semantics checker. The request interpreter and semantics checker are currently under development. Further details on these can be found in [Bob84]. Other components shown in the figure 4.1 are briefly described below.

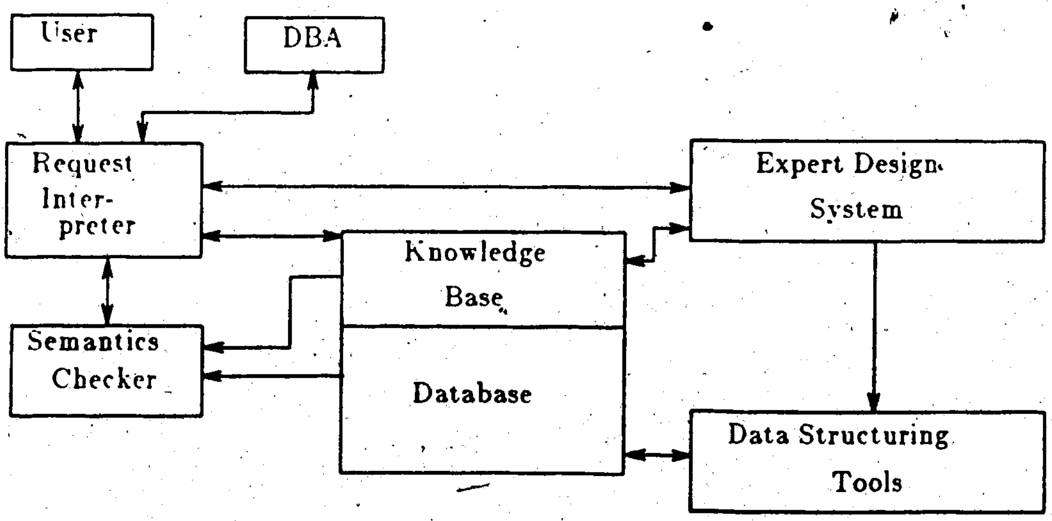

Figure 4.1 Semilattice Database Management System

4.2.1. Knowledge Base or Metadata

The knowledge base or metadata essentially consists of semilattice relations which store information about actual data in a semilattice database system. Storing this information in the form of semilattice relations has the following advantages:

(1) The same routines can be used for accessing data in the knowledge base as well as the actual database.

(2) The request interpreter and semantics checker can be used for manipulating information in the knowledge base.

A knowledge base should include the following information about the database:

(a) Logical structure of each relation

(b) Present physical organization of data in each relation

(c) Meaning of data stored in each relation. Meanings are stored in simple textual form and are for human use only.

(d) Various constraints upon the data, e.g. various functional, multivalued and join dependencies etc.

(e) A set of data usage patterns and their frequencies of occurrence.

(f) A set of pending data reorganization decisions

No doubt information would continue to be added or deleted as the system undergoes further development.

4.2.2. Database

The database consists of the actual data stored by the user. The data is stored in the form of semilattice relations whose conceptual structure is visible to a user. Data in these relations is organized in the form of three kinds of C-B^*-tree files as described in the previous chapter. In order to provide a large degree of freedom in organizing data using the C-B^*-tree structure, the records for a relation in a decoding file are conceptually divided into what we call **partitions.** The goal behind the concept of partitioning a decoding file is explained below.

4.2.2.1. Partitions within a Decoding File

Records in a decoding C-B^*-tree file associated with a relation are conceptually divided into partitions. Records of a decoding file belonging to two different partitions might differ in their physical structure or the optional information they carry. Physical structures of records in all partitions associated with a decoding file are described by a relation in the knowledge base.

The concept of creating conceptual partitions has the following advantages:

(1) Records in the same decoding file can be structured differently.

(2) Incremental reorganization of data in a decoding file is possible. This is essential for a very large database, one which cannot be taken off-line for reorganization.

(3) It is possible to describe some exceptional conditions associated with a certain tuple by using a different record format.

4.2.3. Data Structuring Tools

These are low level procedures for storing, retrieving, updating and restructuring data in a semilattice database system. These are accessed via The Expert Design System.

4.2.4. Expert Design System

The expert design system (E.D.S.) contains algorithms for monitoring the database usage, storing the relevant statistics in the knowledge base and taking decisions regarding reorganization of data. Decisions regarding reorganization of data are stored in the knowledge base. The database administrator may from time to time invoke data structuring tools for executing the reorganizational decisions. Although there are many kinds of data reorganizational issues that should be resolved in a real database environment, we choose the following two problems as a first cut at the problem of adaptive reorganization of data in a semilattice database environment:

(1) Selection of domains of a relation for which secondary index files should exist.

(2) Selection of data that should be redundantly copied into a relation from other relations in the database in order to improve response time.

The two issues as well as their importance in a semilattice database environment are explained below.

4.2.4.1. Selection of Secondary Indexes

Index selection is one of the most complex problems in the entire database design process. The index selection problem can be defined as the problem of choosing primary keys, secondary keys, and index configurations in order to effect a reasonably good database system performance for all types of applications. The index configuration for our semilattice database management system has already been described in Chapter 3 as the secondary C-B^*-tree file. Also, in a semilattice database environment, no primary keys are assumed to exist and one is allowed to create and make proper use of any set of secondary indexes with equal ease. Therefore, the problem entirely reduces to one of choosing secondary indexes.

Clearly, the presence of a secondary index file for a particular domain can improve the execution of many queries that reference that domain; on the other hand, maintenance of such an index has costs that slow down the performance of database updates, insertions and deletions. Roughly speaking, a domain that is much more frequently referenced than modified is a good candidate for index maintenance. The choice of which domains (if any) to select for this purpose should be done with care; a good choice can significantly improve the performance of the system, while a bad selection can seriously degrade it. One goal of our expert design system should be to make a good choice of those domains for which secondary indexes are maintained, based on how the data is actually used.

In optimal secondary key selection one chooses a set of domains that optimize the database system performance. Performance is measured in terms of the various costs e.g. retrieve and update I/O time, CPU overhead, storage space, or some subset of these costs. In practice however, although some heuristics exist for index selection, there is no efficient algorithm to arrive at an optimal index selection. Comer has shown that no algorithm for this purpose has better than 2^n steps; because it is a potentially difficult problem [Com78].

4.2.4.1.1. Previous Work on Index Selection

A number of researchers have provided analytical solutions to the problem under various kinds of simplifying assumptions. Notable among these are studies by Stonebraker [Sto74], Schkolnick [Sch75] and King [Kin74]. Practical solutions to the problem were investigated by Farley and Schuster [FaS75] and Hammer and Chan [HaC76]. In the study by Farley and Schuster, the domains to be inverted are selected by analyzing a typical set of queries submitted to the system. The work by Hammer and Chan comes closest to our goal of identifying usage patterns on the data and utilizing this knowledge for the purpose of selecting a good set of secondary indexes. The operation of their prototype system can be described as follows. The specification of database interactions, by both interactive users and application programs, is expressed in a non-procedural language. This is first translated into an internal representation made up of calls to system level modules. The language processor has available to it a model of the current state of the database, which contains, among other things, a list of the currently maintained secondary indexes, plus various information about these indexes. Using this information, the language processor can choose the best strategy for processing each database operation in the current environment. Statistics-gathering modules are embedded within the system modules that interpret the object code of the language processor. These mechanisms are used to record data concerning the execution of every database transaction. When the reorganizational component of the system is invoked (which occurs at fixed intervals of time), the statistical information gathered over the preceding interval is combined with statistics from previous intervals and is used to obtain a forecast of the access requirements of the upcoming interval. In addition, a projected assessment of various characteristics of data during the next interval is made. A near optimal set of domains for which indexes should be maintained is then determined heuristically. Optimality here means with respect to total cost, taking into account the expense of index storage and maintenance. This minimal cost is then compared with the projected cost for the existing set of indexes. Database reorganization is performed only if the cost benefit is great enough to cover the reorganization cost as well as the cost of application program retranslation.

In general, any practical heuristic approach for finding a good solution to the index selection problem should include the following three basic steps in its process:

(1) Monitor the frequencies of accesses and updates of various domains

(2) Eliminate bad choices for creating the secondary indexes. Domains that are very infrequently accessed can be safely eliminated. This reduces the size of the problem space.

(3) Select a subset of domains for maintaining secondary indexes that gives a solution of acceptable quality.

4.2.4.2. Level of Redundancy in a Database

Proper clustering of data can make a significant difference in the efficiency of a database system because it takes groups of data that are to be frequently accessed together and allows them to be stored physically in a way that maximizes sequential access and minimizes random access.

According to the semilattice condition for designing the logical structure of a database, if two relation schemes R_1 and R_2 have a common part R_c, then the scheme for the common part R_c must appear in the logical design of the database. The common part R_c then appears in the scheme of R_1 and R_2 as a reference to appropriate tuple(s) in relation R_c. This form of reference to the parts of an object in the logical scheme may require some kind of physical clustering of information or copying information redundantly into a relation in order to improve the performance of the system. For example, in the HVFC database if most of the queries on the ORDER relation require retrieval of information regarding the member who ordered an item then it is beneficial to copy the information regarding the member in the ORDER relation as well. However copying information in this way also has a negative side effect. Every time the MEMBER relation is updated, a corresponding change in the ORDER relation is required as well. Therefore such copying of information should be done with caution. The other reorganizational issue that should be resolved in this context is the number of places a certain piece of information should be copied. For example, in the

Chip database, in the CONNECTION relation, if we want to know the location of connection segments joining a particular pair of pins, then relations LINE and POINT should be accessed. Now the question is whether the value of co-ordinates from the POINT relation should be copied into the LINE relation or the CONNECTION relation, or at both places or at neither. In each case the goal is to maximize system performance.

In general, optimal clustering is difficult to achieve in a complex integrated database where data accessed by different application programs overlap, and tradeoffs between sequential and random access must be evaluated. In a semilattice database environment, if a particular relation refers to n other relations then there are 2^n ways in which information can be redundantly copied into this relation. Similarly, if there is a hierarchy of n relations in a semilattice database scheme, then there are 2^n ways in which information from the relation at root of this hierachy can be copied into other relations in the hierarchy.

4.2.4.2.1. Previous Work on Some Related Problems

Although the problem of redundantly copying data in the above mentioned manner is specifically a problem of the semilattice database management system, similar data partitioning and clustering problems exist with other kinds of database systems. The record clustering problem for hierarchical databases has been studied by Schkolnick [Sch77]. The essential difference between his approach and our problem is that in his problem Schkolnick considers clustering of records without introducing any redundancy. If there are n record types, then his method takes time proportional to n. His approach takes the view that a tree with root x can be regarded as a subtree of a larger tree with root y. Therefore it is possible to develop a heuristic procedure that eliminates non-optimal clusterings in the subtree at x from further consideration in the whole tree at y. The same approach can be applied to a limited degree to network databases. A more general and much more complex methodology for CODASYL record placement and area design using heuristic algorithms, a graph theoretic model and an optimization technique has been investigated but not validated in an operating environment [TeF82].

A dynamic approach for data clustering has been discussed by Belford [Bel80]. In her approach, the usage of database is monitored and a matrix Q is generated. Each column of Q corresponds to a particular attribute. The i^{th} row of Q is a binary vector p_i indicating the set of records retrieved by the query pattern i. A "nearest centroid algorithm" is used for generating query patterns p_1, p_2, ... from the observed queries q_1, q_2, ... These query patterns reflect association among attributes that are often required simultaneously. An attribute may be assigned to several such clusters (and so is stored redundantly). Ordinarily, redundancy of records is avoided as being wasteful. In this case, however, the overall efficiency of the system may be better served by occasional repetition of records. Currently, experiments are being performed by Belford in order to ascertain the impact of certain parameters on the algorithm.

4.3. Conclusion

The purpose of this chapter was to study the practical feasibility of dynamic reorganization of data in a semilattice database system. The results of previous work by other researchers in this direction are encouraging. The design of the C-B^*- tree structure has been made keeping this long term goal in mind. The concept of partitioning a C-B^* structure would allow us to do one of the above mentioned reorganizations (specifically, copying data redundantly) in an incremental fashion. The possibility of incorporating other kinds of dynamic reorganizations in the scheme has been left for future work.

Chapter 5

Directions for Future Work and Conclusion

The work presented in this thesis is a part of the larger effort currently being made at the Department of Computing Science, University of Alberta in order to develop the first prototype database management system based on the semilattice data model. The thesis clearly demonstrates the feasibility of implementing schemes for the physical organization of data and various basic operations needed for a semilattice database system. A single data structure called C-B^*-tree has been proposed for all kinds of basic access operations in the database system. This data structure has been implemented under the 4.2 BSD UNIX operating system running on a VAX 11/780* machine in the department. Schemes have been proposed for generating semilattice codes for tuples and for recycling deallocated codes. A significant side effect of developing the C-B^*-tree structure is the development of a new strategy for redistributing records among neighboring pages in case of a page underflow or overflow in a variable-length-record B-tree. Experiments have shown that this strategy works better than previously known strategies for the same purpose. Finally, a preliminary study has been done in order to explore the possibility of dynamically reorganizing data in a semilattice database system.

However, it has not been possible to deal with some of the issues to our full satisfaction and therefore the work of this thesis can be extended in the following ways:

First, it might be necessary to extend or modify the proposed access methods for the semilattice DBMS system. In order to do this, however, it would be worth while to wait for the insights gained by running the prototype system [Cha81, SEP77, Sto80].

Second, the processing of range queries and neighborhood search queries is not handled very satisfactorily. One reason for this is that semilattice codes do not

* VAX is a trade mark of Digital Equipment Corporation.

preserve the natural order of values of any of the components they represent. The situation may be somewhat improved by generating codes that preserve the natural order on a single attribute or a combination of attributes. Some such techniques may be found in [Sor78].

Third, in our data structure, combined indexing has only been implemented for semilattice codes. However, the implementation of combined indexes in general was not considered. Sometimes the presence of such indexes improves the performance of the system [TeF82]. Also, we did not consider implementing higher level semilattice operations directly. Some related work in this direction for relational databases has been described by Gotleib[Got75] and Blasgen and Eswaran[BlE77].

Fourth, the CS pagination strategy presented in the thesis shows about 5% to 10% improvement over the MS strategy. It seems that the extent of improvement depends upon factors like blocking, distribution of possible key lengths etc; however the nature of this dependence was not determined. In order to do this, many more experiments need to be done on our implemented C-B^*-tree structure.

Finally, completing the design of the proposed adaptive data structuring system is a major task. The study by Blasgen and Eswaran during the design of System R [BlE77] shows that in relational database systems, physical clustering of logically adjacent items is a critical performance parameter. In the absence of such clusterings, methods that depend upon sorting the records themselves seem to be the algorithm of choice. Various efforts by researchers for achieving these ends dynamically were described in the previous chapter. The algorithms for this depend upon the data structures and access paths available for processing a query and the utilization of these paths by the query decomposition and optimization algorithms.

References

[AbB84] S. Abiteboul and N. Bidoit, Non-first Normal Form Relations to Represent Hierarchically Organized Data, *3rd ACM SIGACT News-SIGMOD conf. on Principles of Database Systems*, 1984, 191-200.

[Arm84] W. W. Armstrong, A Semilattice Database System, *(Unpublished) Dept. of Comp. Sc., Univ. of Alberta*, 1984.

[Ast76] M. M. Astrahan , System R: A Relational Approach to Data Management, *ACM Trans. Database Systems 1*, 2 (1976), 97-137.

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman, Compilers, Principles, Techniques, and Tools, *Addision-Wesley Publishing Company*, 1986.

[Atr80] S. Atre, Data Base: Structured Techniques for Design, Performance and Management with Case Studies, *John Wiley & Sons*, 1980.

[BaM72] R. Bayer and E. McCreight, Organization and Maintenance of Large Order Indexes, *Acta Informatica 1*, 3 (1972), 173-189.

[BaU77] R. Bayer and K. Unterauer, Prefix B-trees, *ACM-Trans. Databdse Syst. 2*, 1 (March 1977), 11-26.

[Bel75] G. Belford, Dynamic Data Clustering and Partitioning, *Univ. of Illinois at Urbana-Champaign*, 1975.

[Ben75] J. L. Bentley, Multidimensional Search Trees Used for Associative Searching, *Comm. ACM 18*, 9 (Sept. 1975), 509-517.

[Bla81] M. W. Blasgen , System R: An Architectural Overview, *IBM System J. 20*, 1 (1981), P 41-62.

[BlE77] M. W. Blasgen and K. P. Eswaran, Storage Accesses in Relational Databases, *IBM System J. 4*, (1977), 363-377.

[Bob84] K. Bobey, The Semilattice Data Model: Conceptual Level General Specification, *(Unpublished) Dept. of Comp. Sc., Univ. of Alberta*, 1984.

[Car75] A. F. Cardenas, Analysis and Performance of Inverted Data Base Structures, *Comm. ACM 18*, 5 (1975), 253-264.

[Cha81] D. D. Chamberlin , A History and Evaluation of System R, *Comm. ACM 24*, 10 (1981), 632-646.

[ChF79] J. M. Chang and K. S. Fu, Extended K-d-tree Database Organization: A Dynamic Multiattribute Clustering Method, *Proc. 3rd COMPSAC Conf., Chicago*, Nov. 1979, 39-44.

[Com78] D. Comer, The Difficulty of Optimum Index Selection, *ACM Trans. Database Systems 3*, 4 (Dec. 1978), 440-445.

[Com79] D. Comer, The Ubiquitous B-Tree, *Computing Surveys 11*, 2 (June 1979), 121-137

[Dat77] C. J. Date, An Introduction to Database Systems, *Addison-Wesley Publishing Company*, Second Edition, 1977.

[Dum56] A. I. Dumey, Indexing for Rapid Random Access Memory Systems, *Computers and Automation 5*, 12 (Dec. 1956), 6-9.

[FNP79] R. Fagin, J. Nievergelt, N. Pippenger and H. R. Stron endible Hashing- A Fast Access Method for Dynamic Files, *ACM Trans.* *abase Systems 4*, 2 (Sept. 1979), 315-344.

[FaS75] J. H. G. Farley and S. A. Schuster, Query Execution and Index Selection for Relational Databases, *Tech. Rep. CSRG-53, University of Toronto*, March 1975.

[FiB75] R. A. Finkel and J. L. Bentley, Quad Trees - A Data Structure for Retrieval on Composite Keys, *Acta Informatica 4*, (1975), 1-10.

[Got75] L. Gotleib, Computing Joins of Relations, *Proc. ACM-SIGMOD conf.*, San Jose, CA, May 1975, 53-63

[HaC76] M. Hammer and A. Chan, Index Selection in a Self Adaptive Data Base Management System, *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, 1976, 1-8.

[HaN79] M. Hammer and B. Niamir, A Heuristic Approach to Attribute Partitioning, *Proc. of ACM-SIGMOD Intl. Conf. on Management of Data*, 1979, 93-100.

[Har84] M. Hardwick, Extending the Relational Data Model for Design Applications, *Proc. 21st Design Automation Conference*, Albuquerque, 1984, 110-116.

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

[HsH70] D. Hsiao and F. Harary, A Formal System for Information Retrieval from Files, *Comm. ACM 13*, 2 (1970), 67-73.

[IBM66] IBM, Introduction to IBM Direct Access Storage Devices and Organization Methods, *GC 20-1649-06*, 1966.

[Kin74] W. F. King, On the Selection of Indices for a File, *IBM Tech. Rep. RJ1341*, San Jose, CA, 1974.

[KKA84] H. Kitigawa, T. L. Kunii, M. Azuma and S. Misaki, Formgraphics: A Form-based Graphics Architecture Providing a Database Workbench, *Computer Graphics and Applications, IEEE*, June 1984, 38-56.

[Kno71] G. D. Knott, Expandable Open Addressing Hash Table Storage and Retrieval, *Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control*, 1971, 186-206.

[Knu73] D. Knuth, The Art of Computer Programming, *Addison-Wesley Pub. Co. 3*, (1973).

[Kri73] J. D. Krinos, Interaction Statistics from a Database Management System, *AFIPS Press 42*, (1973), 283-290.

[Lar80] P. Larson, Linear Hashing with Partial Expansions, *Proc. Intl. Conf. on Very Large Data Bases*, Montreal, Oct. 1980, 224-232.

[LeW80] D. T. Lee and C. K. Wong, Quintary Trees: A File Structure for Multidimensional Database Systems, *ACM Trans. Database Systems 5*, 3 (Sept. 1980), 339-353.

[Lit80] W. Litwin, Linear Hashing: A New Tool for File and Table Addressing, *Proc. 6th Intl Conf. on Very Large Data Bases*, Oct. 1980, 212-223.

[LiY77] J. H. Liou and S. B. Yao, Multidimensional Clustering for Data Base Organizations, *Info. System 2*, 4 (1977), 187-198.

[Lor74] R. A. Lorie, XRM - An Extended Relational Memory, *Report No. G320-2096, IBM Cambridge Scientific Center, Cambridge*, 1974.

[McC77] E. McCreight, Pagination of B-trees with Variable Length Records, *Comm. ACM 20*, 9 (Sept. 1977), 670-674.

[OlJ76] N. N. Oliver and J. J. Joyce, Performance Monitor for a Relational Information System, *Proc. of the 1976 ACM Conference*, 1976, 329-333.

[MeO81] T. H. Merrett and E. Otoo, Multidimensional Paging for Associative Searching, *McGill Univ. Tech. Rept.*, May 1981.

[Ols69] C. A. Olson, Random Access File Organization for Indirectly Addressed Records, *Proc. ACM National Conf. 24*, (1969), 539-549.

[Rob81] J. T. Robinson, The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes, *Proc. ACM-SIGMOD*, Ann Arbor, April

1981, 10-18.

[RoL74] J. B. Rothnie and T. Lozano, Attribute File Organization in a Paged Memory Environment, *Comm. ACM 17*, 2 (Feb. 1974), 63-69.

[Sch75] M. Schkolnick, The Optimal Selection of Secondary Indices for Files, *Info. Syst. 1*, (1975), 141-146.

[Sch77] M. Schkolnick, A Clustering Algorithm for Hierarchical Structures, *ACM Trans. Database Syst. 2*, 1 (1977), 27-74.

[Sev72] D. G. Severance, Some Generalized Modeling Structures for Use in Design of File Organization, *The University of Michigan*, Ann Arbor, 1972.

[Sor78] P. G. Sorenson, Key-to-Address Transformation Techniques, *Infor 16*, 1 (Feb. 1978), 1-34.

[StG74] J. Steuert and J. Goldman, The Relational Data Management System: A Perspective, *ACM-SIGMOD Workshop on Data Description, Access and Control*, 1974, 295-320.

[Sto74] M. Stonebraker, The Choice of Partial Inversions and Combined Indices, *Intl. J. Compt. Inf. Sci. 3*, 2 (1974), 167-188.

[SWK76] M. Stonebraker, E. Wong, P. Kreps and G. Held, The Design and Implementation of INGRES, *ACM Trans. Database Systems 1*, 3 (1976), 189-222.

[Sto80] M. Stonebraker, Retrospection on a Database System, *ACM Trans. Database Systems 5*, 2 (1980), 225-240.

[TeD76] T. J. Teorey and K. S. Das, Application of an Analytical Model to Evaluate Storage Structures, *Proc. of the Intl. ACM-SIGMOD Conf. on Management of Data*, 1976, 9-20.

[TeF82] T. J. Teorey and J. P. Fry, Design of Database Structures, *Prentice-Hall, Inc.*, 1982.

[Ten81] R. D. Tennent, Principles of Programming Languages, *Prentice Hall International*, 1981.

[Tod76] S. Todd, The Peterlee Relational Test Vehicle - A System Overview, *IBM System J. 15*, 4 (1976), 285-307.

[TrS76] J. P. Trembley and P. G. Sorenson, An Introduction to Data Structures with Applications, *McGraw Hill, New York*, 1976.

[Ull82] J. D. Ullman, Principles of Database Systems, *Computer Science Press*, Second Edition, 1982.

[Whi74] V. K. M. Whitney, Relational Data Management Implementation Techniques. *ACM-SIGMOD Workshop on Data Description, Access and Control*, 1974, 321-348.

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

[YaM75] S. B. Yao and A. ~~G. Merten~~, Selection of File Organization Using an Analytical Model *Proc. of the First Intl. Conf. on Very Large Data Bases*, 1975, 255-267.

[Yao77] S. B. Yao, An Attribute Based Model for Database Access Cost Analysis. *ACM Trans. Database Systems 2*, 1 (1977), 45-67.

[Yu85] C. T. Yu, Adaptive Record Clustering, *ACM Trans. Database Systems 10*, 2 (1985), 180-204.

Appendix A1

Appendix

The database schemes for the Chip database and HVFC database were given in Chapter 2 (section 2.2.2). An instance of database based on each of these schemes is presented here. The name of an attribute or tuple whose value is given in coded form is indicated with a slash (/) character. For example, A/ means codes for tuples of the relation A; similarly, A,..M/N,...,P stands for code for attributes (A, ..., M) relative to values of attributes (N, ..., P).

1.1. An Instance of the Chip Database

The database scheme presented in Chapter 2 can be used to describe the functional diagrams of simple electronic components. For example figure-A.1 illustrates the making of a 4-AND gate in terms of three 2-AND gates.

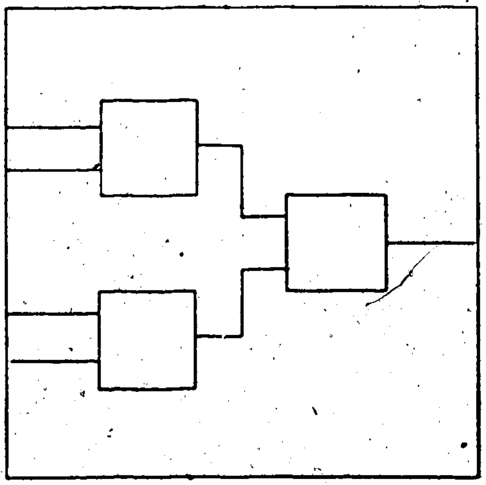

Figure A1.1 Structure of a 4-AND Gate

The instance of the chip database given in Table A-I through A-VII describes the topology of this circuit. Tuple-schemes for POLYGON, CONNECTION, BAS_ENTITY and COM_ENTITY relations contain one or more attributes with domains whose members are sets or sequences. Therefore tuples of these relations are assigned absolute codes. Relative codes are generated for the tuples in the other five relations. In storing the relation ENTITY the null tag is used to indicate that the reffered to tuple belongs to the BAS_ENTITY relation and a tag value of 0 is used for reffering to a tuple in the COM_ENTITY relation.

x:real	y:real	POINT/
0	0	0
0	3	1
0	7	2
0	10	3
0	13	4
0	17	5
0	33	6
0	37	7
0	50	8
10	0	9
10	5	10
10	10	11
10	13	12
10	17	13
10	30	14
10	33	15
10	37	16
20	15	17
20	35	18
25	15	19
25	23	20
25	27	21
25	35	22
30	20	23
30	23	24
30	27	25
40	25	26
50	0	27
50	25	28
50	50	29

Table A1.1 The Relation "POINT"

point1: point	point2: point	LINE/
0	3	0:3
3	11	3:11
11	9	11:9
9	0	9:0
0	8	0:8
8	29	8:29
29	27	29:27
27	0	27:0
4	12	4:12
5	13	5:13
6	15	6:15
7	16	7:16
17	19	17:19
18	22	18:22
19	20	19:20
21	22	21:22
20	24	20:24
21	25	21:25
26	28	26:28

Table A1.2 The Relation "LINE"

polygon:{line}	POLYGON/
0:3, 3:11, 11:9, 9:0	0
0:8, 8:29, 29:27, 27:0	1

Table A1.3 The Relation "POLYGON"

entid: integer	instance: integer	place: point	(entid, instance)/ place	BLOCK/
0	1	11	0	(11:0)
0	2	14	0	(14:0)
0	3	23	0	(23,0)
0:0	1	0	0	(0:0)

Table A1.4 The Relation "BLOCK"

entid: integer	class: binary	pin#: integer	place: point	(entid,class, pin#)/place	PIN/
0	0	1	1	0	1:0
0	0	2	2	0	2:0
0	1	3	10	0	10:0
0:0	0	1	4	0	4:0
0:0	0	2	5	0	5:0
0:0	0	3	6	0	6:0
0:0	0	4	7	0	7:0
0:0	1	5	28	0	28:0

Table A1.5 The Relation "PIN"

block1: block	pin1: pin	block2: block	pin2: pin	con-seg: <line>	CONNECTION/
0:0	4:0	11:0	1:0	4:12	0
0:0	5:0	11:0	2:0	5:13	1
0:0	6:0	14:0	1:0	6:15	2
0:0	7:0	14:0	2:0	7:16	3
11:0	10:0	23:0	1:0	(17:19, 19:20, 20:24)	4
14:0	10:0	23:0	2:0	(18:22, 22:21, 21:25)	5
23:0	10:0	0:0	6:0	26:28	6

Table A1.6 The Relation "CONNECTION"

entid: integer	shape: polygon	ports: {pin}	BAS_ENTITY/
100	0	(1:0,2:0,10:0)	0

Table A1.7 The Relation "BAS_ENTITY"

entid: integer	shape: polygon	components: {block}	ports: {pin}	inter_con: {connection}	COM_ENTITY/
200	1	(11:0,14:0, 23:0)	(4:0,5:0,6:0, 7:0,28:0)	0,1,2,3, 4,5,6)	0

Table A1.8 The Relation "COM_ENTITY"

(b_ent:bas_ent, c_ent:com_ent)	ENTITY/
0 0:0	0 0:0

Table A1.9 The Relation "ENTITY"

1.2. An Instance of the HVFC Database

Here again, relative codes are assigned to the tuples of the MEMBER, ORDER and PRICED_ITEM relations whereas an absolute code is assigned to each tuple in the SUPPLIER relation.

name: char(30)	address: char(50)	balance: real	MEMBER/
Brooks, B.	7, Apple Rd.	+10.50	0
Field, W.	43, Cherry Ln.	0	1
Robin, R.	12, Heather St.	-123.45	2
Hart, W.	65, Lark Rd.	-43.30	3

Table A1.10 The Relation "MEMBER"

item: char(20)	price: real	PRICED_ITEM/
Granola	1.29	0
Lettuce	0.89	1
Sf. seeds	1.09	2
Whey	0.70	3
Curds	0.80	4
U. Flour	0.65	5
Granola	1.25	6
Lettuce	0.79	7
Whey	0.79	8
Sf. seeds	1.19	9

Table A1.11 The Relation "PRICED_ITEM"

order_no: integer	purchaser: member	item: priced_item	quantity: integer	(order_no, quantity)/ (purchaser, item)	ORDER/
1	0	0	5	0	0:0:0
2	0	5	10	0	0:5:0
3	2	6	3	0	2:6:0
4	3	3	5	0	3:3:0
5	2	9	2	0	2:9:0
6	2	7	8	0	2:7:0

Table A1.12 The Relation "ORDER"

sname: [illegible]	saddress: char(50)	stock: {priced_item}	SUPPLIER/
Sunshine Produce	16 River St.	(0,1,2)	0
Purity Food Stuff	180 Industrial Rd.	(3,4,5,6)	1
Tasti Supply Co.	17 River St.	(7,8,9)	2

Table A1.13 The Relation "SUPPLIER"