An Approach to Semi-Automatic Physical Database Design and Evolution for Personal Information Systems

Technical Report CRI-85-11
September 19, 1985

Rafiul Ahad and Dennis McLeod

Department of Computer Science

University of Southern California

Los Angeles, CA 90089-0782

Abstract

An approach to allow the physical storage organization of a database system adapt to the prevailing usage pattern with minimum human intervention is presented. The approach presented herein views a conceptual database as a set of logical access paths of arbitrary lengths connecting database objects. Transactions against a database are characterized in terms of the logical access paths they reference. The logical access path reference pattern is monitored while the database is being used. At some user-determined time, this pattern is used to redesign the physical storage configuration. The goal of the physical (re)design process is to directly represent the most frequently used logical access paths at the storage level, thereby reducing the time required to construct them at run time. A physical storage structure called the List-based storage structure is used to represent logical access paths. The storage structure provides restructuring operations that facilitate incremental physical (re)design. This paper describes an experimental prototype system (ADAMS), that incorporates the above ideas. The description of ADAMS includes the data model used for capturing the logical access path reference pattern, the model for specifying the physical database (i.e., the List-based storage structure), the binding of logical access paths to physical access paths, and the physical design methodology. Performance studies using simulation, to compare the technique presented herein with a conventional database system, are discussed.

1. Introduction

With the advent of personal computers and work stations, a new kind of data-intensive application environment has emerged. Important characteristics of this environment include [25]:

- The end-user is familiar with the application environment but does not usually have database expertise.
- The environment has no "database administrator".
- The information usage pattern is dynamic.
- The amount of structural information (e.g., kinds of data, kinds of inter-relationships, etc.) is large relative to the size of the database.

This characterization applies to environments ranging from personal information management to engineering design environments. The characteristics of this environment, which is the target of the research described in this paper, warrant investigation of the functions, techniques, and structures of a database system for such an environment.

In order to give a detailed description of the goals of the research, the terminology used in this paper is first given. A *database* (DB) is a two-tuple $<S,D>$, where D is a time-varying collection of data items, and S is a schema describing the structure, constraints and other relevant metainformation about the data in D. Under this definition, two databases are considered to be the same if their schemata are the same. A *data model* (DM) is a tool for defining and manipulating a database, and thus provides facilities to define the structure (schema) of a database; to create, access, and manipulate data; and facilities to specify constraints on the structures and operations. A *query* against a database is a sequence of operations intended to isolate or modify some meaningful subset of the database. A *transaction* against a database is a sequence of queries, perhaps including other non-query processing, intended to retrieve or change information from/in the database. A database, plus all the transactions defined to run against it, is called a *database environment* (DBE).

DBs, queries, transactions, and DBEs can be qualified as "conceptual", "intermediate", or "physical" to indicate the level of abstraction; the conceptual level is the closest to the real world, and the physical level models the storage and access structures. A *database system* (DBS) is a collection of DBEs of different levels of abstraction, and the specification of the mappings from one DBE to another. In a database system, a conceptual database environment is first *bound* to a physical database environment. Binding is the process of translating structures, constraints, and operations of one DBE to another. Binding is done by a database management system (DBMS) - a system that implements the tools associated with a database system. In general, a conceptual database environment can be bound to one of many physical database environments, each of which exhibits different performance characteristics for the same usage pattern depending on the distribution of data items on the physical medium and the presence of auxiliary structures. Auxiliary structures, such as access paths, are constructs of a physical database, designed to enhance performance, and usually have no counterparts in the conceptual database. The process of finding an optimal[1] physical database environment for a conceptual database environment is termed *physical database design*.

Physical database design requires many input parameters, the most important of which is the usage pattern of the database system. Usage patterns are generally not stable [29, 33]. Some of the reasons for the instability are: an inaccurate usage pattern used for the initial design of a physical database; diversifying use of a database system; rule, regulation, and policy changes; and seasonal changes in information requirements. If the usage pattern changes during the course of the database use, then the current physical database may no longer be optimal, resulting in performance deterioration. To keep the performance at an acceptable level, physical database redesign is required to find a more suitable physical database supporting the same conceptual database. This is known as the evolution of a physical database.

With most current database systems, the initial physical database design and subsequent evolution of a physical database are handled by the database administrator

[1]The term "optimal" does not necessarily mean the best. However, the term is firmly established in the literature to mean the best under given constraints.

(DBA) or the users. Since the physical database evolves according to the dynamics of the usage pattern, to maintain an optimal physical database, the DBA must monitor the usage pattern and assess the appropriateness of the physical database instance to the usage pattern. If the physical database instance is found to be inappropriate for the current usage pattern, the DBA or the user must draw upon his/her knowledge to redesign the physical database. If the redesign produces a different physical database instance, the DBA or the user must transform the existing instance into the new instance making sure that the set of facts represented by the new physical database instance is the same as the old one. Further, unless complete data independence is provided by a database system, the DBA or the user must ensure that all the applications remain valid with respect to the new physical database instance each time a change is made to the physical structures. All these tasks require substantial expertise and human resources.

The principal goal of this research is to develop concepts and techniques for a database management system for the target environment. This goal is divided into two sub-goals, the first of which is to incorporate a fair amount of database design knowledge into a DBMS so that it can adapt to the prevailing usage pattern with minimum human intervention. This requires the DBMS to automatically monitor the usage pattern, design a physical database based on the usage pattern, and reorganize the current physical database. The second sub-goal is to utilize a novel physical storage structure and an associated physical design methodology that are suitable for the target environment.

Automatic physical database evolution has of course been the focus of other research. Although the principal goal of the work in this area is to match the storage structure of a database system to the expected usage pattern, in its most general setting, the problem is undecidable; that is if response sets to the queries in a usage pattern are stored physically, then the problem of finding the storage structure for a given query is equivalent to the problem of deciding whether two queries are the same, the latter of which is undecidable. Thus the approach taken in previous work is to simplify a query by restricting it to be a single-variable (involving one relation) query and/or by restricting the syntax of the predicates in the query.

In [13, 35, 36] a self-organizing database management system is described; and [19] presents a self-adaptive support system for multiple queries. The common characteristics of these two systems are that they deal with the relational data model, they maintain files corresponding to frequently used queries thus exploiting controlled redundancy to enhance performance, and they allow a database to be in inconsistent state; i.e., delayed updates of data stored redundantly are permitted. The self-adaptive database management system delineated in [15, 8] differs from the above two systems in that it does not maintain replicated data and it focuses on dynamic selection of indexes for a relational database.

The system described in this paper supports various user-level data models but for illustration we will use the relational data model [9]. In the context of the relational data model, the target environment has a large number of relations. This means that an average query would request data from several relations, requiring a DBMS to perform join operations on these relations. Since the cost of evaluating a join operation is typically high, the performance bottleneck for the target environment lies in the time for performing the join operation. Thus restricting queries to reference a single relation is too unrealistic an assumption. Therefore we have taken an approach that does not place any restrictions on the queries. Queries are decomposed into a sequence of manipulation of logical access paths and a heuristic procedure is used to find the appropriate physical structure(s) for each logical access path. The method employed, phrased in terms of the relational data model terminology, is to partition the attributes of relations and to redundantly represent frequently joined relations in a pre-joined (outer join) form, and store it as an un-normalized (i.e., non-first-normal-form) relation. Hence the need for the novel physical storage structure and the design methodology.

To address these goals, an architecture of an adaptable database system, and an experimental prototype DBMS called the *Adaptable Database Management System*(ADAMS) have been developed. The current version of ADAMS was developed on a UNIX system using the programming language C. About 8000 lines of code were written to implement the routines needed to demonstrate the feasibility of the system. ADAMS is not a fully-automated system. A user of ADAMS is assumed to be capable of determining the redesign time. The determination of redesign time, if delegated to a database system, entails an excessive amount of computation time. The task is best performed by the user in the form of a complaint of unsatisfactory performance reported to the system. The system can initiate the physical design process in response to the complaint. The user is also expected to assist in the formation of a usage pattern for the immediate future. Projection of the usage pattern into the future requires the knowledge of the dynamics of the application environment which, in most cases, is unpredictable. Since no satisfactory mathematical model exists to predict the usage pattern, the user's knowledge of the application environment plays an important role in deciding the usage pattern on which the physical database design is based.

It should be noted that ADAMS is not designed to outperform a database system tuned by a database expert. It is rather intended to provide reasonable performance over an extended usage period. Thus it is expected to provide better performance for an environment where the database is never tuned manually. The purpose of the experimental implementation of ADAMS is to test the feasibility of the principles to support semi-automatic evolution of physical databases. Many useful features of a database system such as concurrency control, authorization, backup and recovery, and the like are not included in the implementation of ADAMS as they are not central issues in this research.

The remainder of this paper is organized as follows. In section 2, an overview of ADAMS is presented. Sections 3 through 6 describe individual components of the architecture of ADAMS. Simulation results are presented in section 8 and concluding remarks are made in section 9.

2. Overview of the Approach

Before presenting an overview of ADAMS, the terms and concepts required to understand the system are first explained. We assume that information about a real-world application is modeled in terms of objects and relationships among objects. Objects are categorized via types; each type contains objects possessing similar relationships or behavior. Relationships among objects are conceptually represented as *access paths*; an access path is recursively defined as a non-empty set of 2-tuples $\{<x_i,Y_i>\}$ where each x_i is an object of some type $\mathcal{X}$ and each Y_i is either a set of objects of type $\mathcal{Y}$, or an access path[2]. An access path A is called a *full* access path if $\forall <x_i,Y_i> \in A$, Y_i is either a non-empty set of objects or a full access path. A is called a maximum access path with respect to $\mathcal{X}$ if $\{x|<x,Y> \in A\} = \mathcal{X}$ at all times. For an access path $A = \{<x_i,Y_i>\}$, if $Y_i = \{y_{i1},...,y_{in}\}$, $n \geq 1$, and each y_{ij} is an object, then A is said to be of length one; if each Y_i is an access path, then the length of A is one plus the longest of the lengths of Y_i's. An access path relates objects of type $\mathcal{X}$ to objects of type $\mathcal{Y}$ and is given a name to reflect the semantics of the relationship. The objects of $\mathcal{X}$ are called the *source* objects and the objects of $\mathcal{Y}$ are called the *destination* objects. For example, the access path $<x,\{<y_1,\{z_1,z_2\}>,<y_2,\{z_3,z_4\}>\}>$ is of length two, and relates the source object x to the destination objects z_1, z_2, z_3, and z_4. The length of this access path is two.

For example, a binary relation R(A,B) of the relational data model is an access path from objects in domain A to those in domain B. This access path is of length one, since it does not involve intermediate objects to represent the relationship. If S(B,C) were another relation, then the relation resulting from joining R and S on B would be an access path from A to C of length two. In terms of a functional data model an access path is a single-argument function or a function resulting from composition of single-argument functions.

The architecture of ADAMS is an extension of the database system architecture proposed by ANSI/SPARC [38, 42]. The portion of the ANSI/SPARC architecture that is extended is shown in figure 2-1. Although the ANSI/SPARC architecture does not explicitly state the requirements for the modeling and manipulation constructs of the conceptual model, to increase programmers' productivity, a high-level non-procedural data manipulation language is desired for a conceptual model [11]. In order to evaluate a query expressed in a high-level non-procedural language, it must eventually be

[2]The notations used here is as follows: script letters denote types, a lower case letter x means an object of type $\mathcal{X}$, an upper case letter X means a subset of type $\mathcal{X}$.

transformed into a procedural query involving access paths. With existing systems, this transformation includes query optimization: a process of choosing minimum-cost access paths based on the currently maintained storage and access structures. In figure 2-1, the conceptual-to-physical mapping supports this transformation.

The usage pattern of a database can be gathered by monitoring the access path reference pattern. In the existing systems, when a non-procedural query is decomposed into a sequence of physical access paths, these access paths are dependent on the physical structures, i.e., the same query may be decomposed into a different sequence of access paths for a different physical structure configuration. Thus, such usage patterns may be skewed and may not adequately reflect the actual access path reference pattern. In order to gather usage pattern information consisting of physical-structure-independent access paths, an intermediate level between the conceptual and the internal levels is desirable. Such a model should represent the logical data as a set of logical access paths and should provide constructs to manipulate individual logical access paths so that it is possible to monitor individual logical access path usage patterns. This implies that data manipulation is procedural and thus the model is not suitable for a conceptual model. In ADAMS, such a model is called the intermediate model.

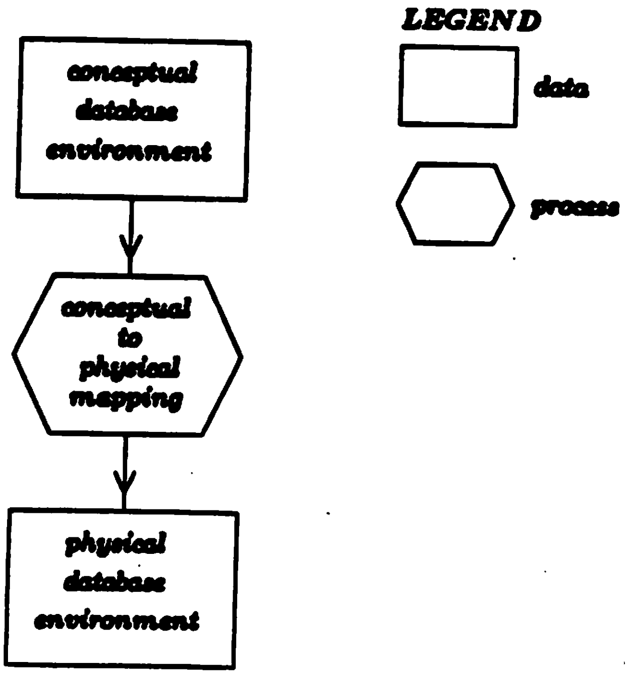
LEGEND
data
process
conceptual database environment
conceptual to physical mapping
physical database environment

Figure 2-1: Portion of Conventional Architecture

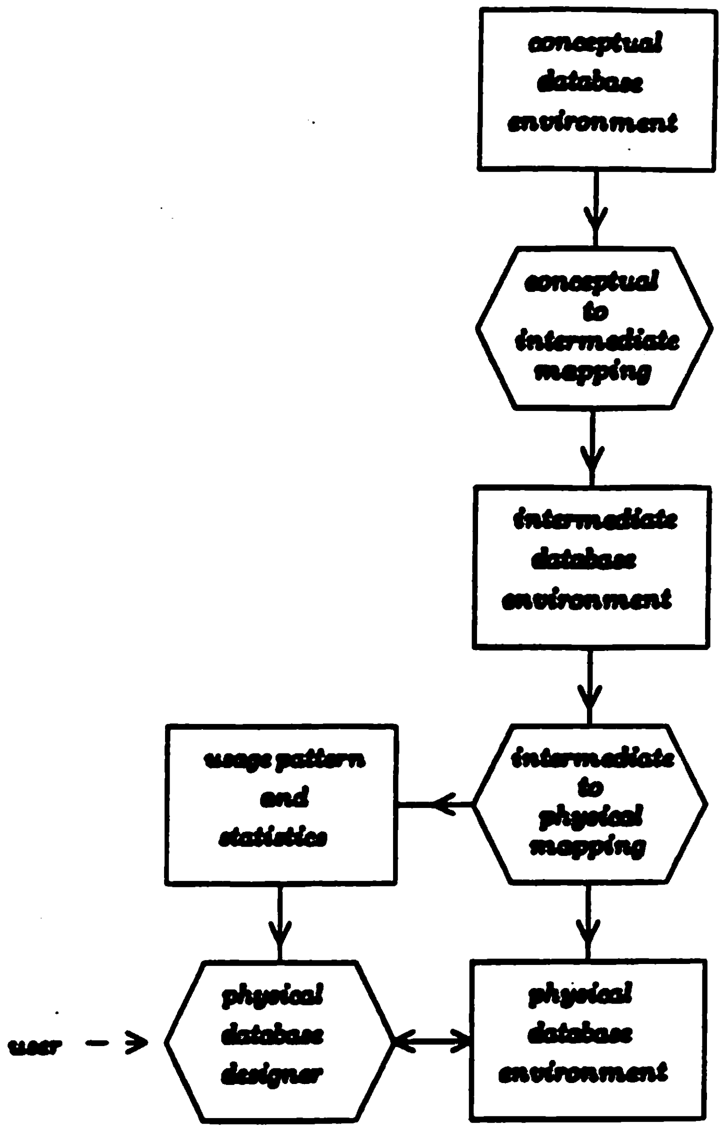
conceptual database environment
conceptual to intermediate mapping
intermediate database environment
usage pattern and statistics
intermediate to physical mapping
user
physical database designer
physical database environment

Figure 2-2: Architecture of ADAMS

In the architecture of ADAMS, a conceptual database environment is first bound to an intermediate database environment. The intermediate data model is an object-oriented binary data model whose modeling constructs are *objects*, and *maps*; maps represent relationships among objects. Objects having similar properties are grouped into *types*, and maps representing a relationship between objects of one type to another, are grouped into a *map type*. A map type is a slightly restricted form of a function used in DAPLEX [34]. A *map set* is a subset of a map type. Thus a map set is an access path of length one, and by composing these map sets one could build up access paths of longer lengths.

Using this method, data manipulation is done as follows. Starting with given sets of objects $S_1,...,S_n$, access paths $A_1,...,A_n$, with S_i the source object set of A_i, are used to retrieve object sets $D_1,...,D_n$. The D_is may be the desired objects, or they may be manipulated using set operations, to obtain $S'_1,...,S'_m$ and the process repeated until desired objects are retrieved. The intermediate model captures all access paths of length one of the conceptual model. For example, in a relational database, if each relation is designed to represent some facts about an object set, then each attribute of the relation can be viewed as a logical access path from the object set to the attribute values. For example, consider the relation STUDENT <S#, SNAME, AGE>. The logical access paths are STUDENT to S#, STUDENT to SNAME, STUDENT to AGE, S# to STUDENT, SNAME to STUDENT, and AGE to STUDENT. The last three access paths are normally called indexes on the attributes. A conceptual transaction could be bound to an intermediate transaction statically, and thus any optimization done during this binding has to be done just once.

If the map types of the intermediate model were to be represented physically, then the performance of a database system would be poor if transactions frequently use long access paths and/or a large number of map types emanating from an object type. This is due to the high cost of composing the access paths of length one to obtain access paths of longer lengths. In order to improve performance, frequently used access paths of length two or longer should also be represented physically. Furthermore, such access paths should be amenable to updates. To represent logical access paths of length one or longer, a storage structure capable of representing nested or non-linear constructs is needed. The storage structure used in the physical database of ADAMS resembles that of the List[3] and is thus called the List-based storage structure [2]. Another reason for choosing the List-based storage structure is to incorporate enough data semantics to allow meaningful restructuring of the physical structures.

[3]The term List (note the upper case "L") is used to refer to the List data structure as defined by Knuth [23].

An intermediate transaction is expressed in a conventional programming language that is enhanced with data manipulation statements. Such a transaction is bound to an equivalent transaction on the corresponding physical database environment by means of a two-phase binding method. The two phases are the compilation phase, and the execution phase. In the compilation phase, the intermediate transaction is first processed and converted into a program in pure host language. This program contains calls to run time routines that perform the following functions:

1. name resolution; this requires the use of a heuristic procedure that searches for the best way to bind a logical access path to one of the available access paths;

2. usage pattern and statistics gathering;

3. optimization, i.e., avoiding multiple reading of the same block;

4. interpretation, i.e., realization of the intermediate level operations.

An intermediate program is compiled using a compiler for the host language and the object code produced from the compilation phase is stored. The second phase, when invoked, executes the program producing the result, and updates the statistical usage data. Statistics are kept for transactions (execution frequency, access paths referenced, etc.), and object and map types (cardinality, space requirement, etc.).

The statistical information is used in the physical database (re)design when the user invokes the *physical database designer* (PDD). The PDD reads the usage pattern and statistics, and interacts with the user to get the expected usage pattern for the immediate future, and in some cases, information regarding the cardinality of relationships.

The PDD uses heuristic procedures to determine how map types should be represented by the underlying List-based structures so that the access paths in all the transactions can be traversed with minimum total number of block accesses. The design procedure consists of two phases. In the first phase, the map types having common domains are partitioned into a number of groups based on the usage pattern. A heuristics is used to form groups to maximize the probability of finding all map types used by a transaction in one group. The second phase adds redundancy to the groups formed in the first phase to reduce the retrieval time of frequently composed map types. During this design process, the PDD produces a series of restructuring operations that are later issued to the physical database environment. The physical database environment reorganizes itself and make appropriate changes to the physical database schema.

3. The Intermediate Model

As described in section 2, the modeling constructs of the intermediate model are objects, object types, maps, and map types. The data manipulation constructs include operations to define and use access paths; insert delete and update maps and map types; and set operations. In this section a detailed description of these constructs is given.

3.1. modeling Constructs of the Intermediate Model

There are two kinds of objects:

- *descriptor objects* are printable values, e.g., ASCII strings, numbers, etc..
- *abstract objects* represent concepts or things, which are not directly displayable.

Objects having similar properties or behavior are grouped into *types*; types can have subtypes which in turn may have subtypes, etc.. A subtype of a type inherits all the map types of its supertype; in addition, a subtype may have map types of its own. Instances of a subtype are also instances of the parent type. A type and all its subtypes form a lattice that represents a generalization hierarchy. An object type X is said to *dominate* an object type Y from the same generalization hierarchy, written $X \succeq Y$, iff $X \supseteq Y$. An object type X is said to be the root of a generalization hierarchy iff for every object type Y of the hierarchy, $X \succeq Y$ at all times.

A map is a 2-tuple <x,Y>, where $x \in \mathcal{X}$, Y is a non-empty subset of $\mathcal{Y}$, and, $\mathcal{X}$ and $\mathcal{Y}$ are two object types, not necessarily distinct. For example, <s1,{"Jones"}>, <s2,{"Smith"}>, <s1,{7920,7454}>, and <s2,{7475}> are all valid maps. For a map <x,Y>, the object x is called the *source object* of the map and the object set Y is called the *destination object set* of the map. The source objects of the four maps shown above are s1, s2, s1, and s2; and the destination object sets are {"Jones"}, {"Smith"}, {7920,7454}, and {7475}. The object type from which a source object is drawn for a map is called the *source object type* of the map, and the object type whose subset appears as a destination object set of a map is called is called the *destination object type* of the map. Thus, if s1 and s2 are objects of type $\mathcal{S}$, then the source object type of the four maps shown above is $\mathcal{S}$. The destination object type of the first two maps could be, for example, $\mathcal{N}$, a type containing the names; and the destination object type of the last two maps could be $\mathcal{P}$, a type containing the phone extension numbers. A map is used to capture a relationship between an object and a set of objects. Each map is given a name to reflect the meaning of the relationship it represents. For example, the first two of the above four maps could be given the name *name*, and the last two could be given the name *phones*. Maps having the same name, the same source object type and the same destination object type are grouped into a *map type*. A map type is thus an access path of length one. The name of the map type is the same as the name of the constituent maps. For example, the first two maps could be grouped into a map type named *name*, and the last two could be grouped into a map type named *phones*. A map type can be considered as a function whose domain is the source object type of the individual maps, and whose range is the power set of the destination object type minus the empty set. Thus the terminology for functions is freely used for map types in the subsequent discussion. For example the domain (i.e., the source object type) of a map type M is denoted Dom(M), and the range (i.e., the destination object type of) of M is denoted Ran(M).

For every map type whose domain is object type $\mathcal{X}$ and whose range is object type $\mathcal{Y}$, there exists a converse map type from $\mathcal{Y}$ to $\mathcal{X}$. In this paper, if the converse of a map type M is not explicitly named then it is referred to as $M. Maps play important role in modeling the data. The properties of map types are used to impose fundamental semantic integrity constraints on the database. The following are the properties of a map type M.

1. *Single-valued*, where each object in the domain is mapped to exactly one object in the range; i.e., M is a single-valued map type implies

$\forall <x,Y> \in M, |Y| = 1.$

2. *Multi-valued*, where each object in the domain is mapped to a set of objects in the range; i.e., M is a multi-valued map type implies $\forall <x,Y> \in M, |Y| \geq 1.$

3. *Total*, where every element in the domain is mapped to some element in the range, i.e., M is a total type implies $\{x \mid <x,Y> \in M\} = Dom(M)$.

4. *Partial*, where some objects in the domain may not be related to any object in the range, i.e., $\{x \mid <x,Y> \in M\} \subseteq Dom(M)$.

A map type M from $\mathcal{X}$ to $\rho(\mathcal{Y})-\phi$ is denoted as

M: X => Y if M is single-valued and partial,

M: X =>| Y if M is single-valued and total,

M: X =≫ Y if M is multi-valued and partial, and

M: X =≫| Y, if m is multi-valued and total.

Notice that a total map type corresponds to a maximal access path of length one. An intermediate schema can be depicted by a directed, labeled, multi-graph. The nodes of the graph represent object types; a shaded node represents a descriptor object type, and a plain node represents an abstract object type. A solid line arc from a node u to a node v represents a map type whose domain is u and whose range is v. A broken line arc from a node u to a node v represents the fact that node u is a subtype of node v, i.e., ISA relationship between types. The properties of a map type are shown by the symbols on the arcs. A single arrowhead means a single-valued map type, a double arrowhead indicates a multi-valued map type, a bar in front of an arrowhead means a total map type, and absence of a bar means a partial map type.

As an example, consider a conceptual schema expressed using the relational data model. Let STUDENTS = <SNO,SNAME>, COURSES = <CNO,UNIT>, and ENROLLMENTS = <SNO,CNO,GRADE> be the three relations representing students, courses and enrollments respectively. The information contained in this conceptual schema could be represented by the intermediate schema shown in fig 3-1.

Notice that the intermediate schema also represents meaningful *joins*; e.g., the arcs between STUDENTS and ENROLLMENTS, ENROLLMENTS and COURSES represent possible joins.

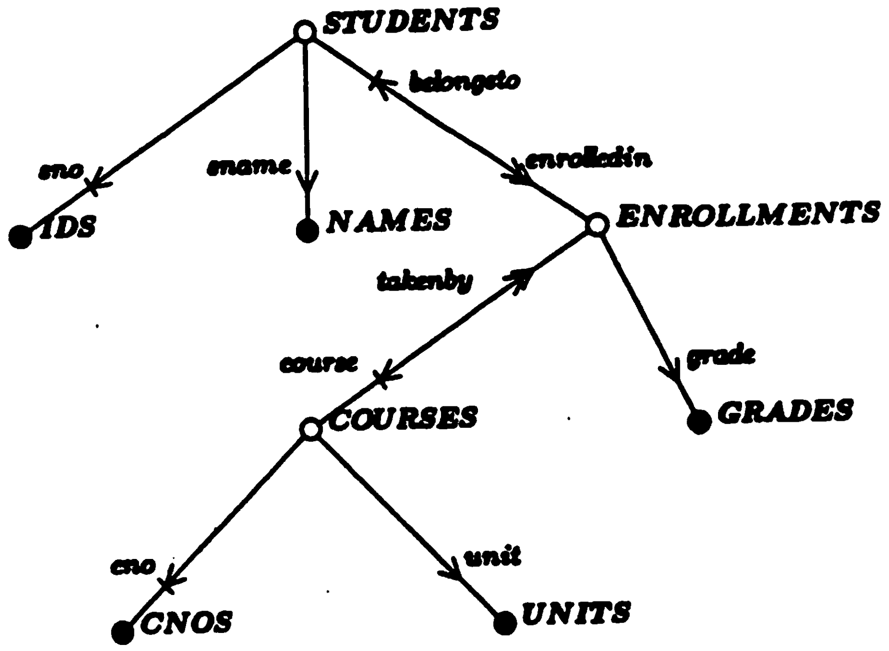

Figure 3-1: Example Intermediate Schema

Map types are logical access paths of length one and are used as building blocks to define access paths of arbitrary lengths. For example, suppose we wish to access the course numbers of the courses that students with the names "Smith" and "Jones" are taking, we could use the access path {"Smith","Jones"} : \$name o enrolledin o course o cno. Here, the operator : is called the selection operator; X : M, where $X \subseteq Dom(M)$ and M is a map type, yields a map type $\{<x,Y> \mid <x,Y> \in M \wedge x \in X\}$. The operator o is called the composition operator[4] ; for any two maps M1 and M2, M1 o M2 yields the map $\{<x,Z> \mid (x \in Dom(M1)) \wedge (Z \subseteq Ran(M2)) \wedge (<x,Y> \in M1) \wedge (Z = \bigcup_{(y \in Y) \wedge (<y,Z'> \in M2)} Z')$.

[4]In actuality, the composition operator is the period symbol. To avoid confusion, the symbol o is used as the composition operator in the text.

Formally, an access path is specified as $R : x_1 \circ \ldots \circ x_n$, $n \geq 1$, where

1. R, the source object set of the access path, is a subset of $Dom(x_1)$,
2. each x_i is a map type,
3. for any x_i and x_j in an access path, if $x_i = x_j$ then $i = j$,
4. for any $x_i \circ x_j$, $Dom(x_j) \succeq Ran(x_i)$.

Notice that according to the above definition, an access path cannot contain the same map type more than once.

3.2. Data Definition and Manipulation Constructs

A transaction of an intermediate database environment is a program written in some host programming language, and containing operations of the intermediate data model in the style of EQUEL programs for INGRES [37]. The operations are described in detail in [3]; these can be broadly classified into the following groups:

- operations to create and destroy objects and object types,
- operations to create, destroy and update maps and map types,
- operations to define access paths,
- operations to define variables to hold objects and object types,
- set operations on object sets, and
- operations to retrieve descriptor objects (printable values) into conventional variables.

If an operation fails, then the *status word* is used to indicate the reason for its failure. The status word has sixteen bits, numbered one to sixteen, the right most bit is numbered one. The status word is cleared before execution of an operation, and if an operation would put the database in an inconsistent state, then the operation is not executed and appropriate bits of the status word are set.

The objects, object types, maps, and map types that are manipulated by a transaction can be divided into two kinds: persistent and temporary. Persistent constructs created by a transaction (using CREATE-xxx[5] operations) become part of a database and can be used by other transactions. The names given to object types and map types during creation are global in a sense that all transactions must refer to these constructs by these names. Temporary constructs (created by DEFINE-xxx operations) are variables that represent objects, object types, maps, and map types; they exist during the life time of a transaction and cease to exist once the transaction terminates. The constructs are referred to by identifiers defined in a program. A program could also contain constants which are specified as follows:

<u>Structure</u>	<u>Format</u>
Descriptor Object	a quoted ASCII string or a number.
Abstract Object Set	$\{x_1,...,x_n\}$ where x_i is a variable of type abstract object.
Descriptor Object Set	$\{v_1,...,v_n\}$ where each v_i is a descriptor object.
map	$<x:y_1,...,y_n>$ where x is variable of type abstract object and y_i is either a variable of type abstract object or a descriptor object.
Map Set	$\{m_1,...,m_n\}$ where m_i is a map.

The following descriptor types are assumed to exist. Here, n is an integer number indicating the size of a value.

<u>Type</u>	<u>Values</u>
P-INTn	$\{i \mid 0 \leq i \leq 2^n\}$
INTn	$\{i \mid -2^{n-1} \leq i \leq 2^{n-1}-1\}$
REALm:n	integer part $\{i \mid -2^{m-1} \leq i \leq 2^{m-1}-1\}$ fraction part $\{i \mid -2^{n-1} \leq i \leq 2^{n-1}-1\}$
F-STRINGn	strings of maximum n ASCII characters
V-STRING	strings of arbitrary length
DATE	a set of date values

The following example shows how the object types STUDENTS and NAMES, and the

[5] Where xxx is one of the strings "AOS", "DOS", "AOT", or "DOT"

maps *sname* and *$sname* of the example database of figure 3-1 can be set up.

```
CREATE-AOT STUDENTS
CREATE-DOT NAMES V-STRING
CREATE-MT sname STUDENTS NAMES S T $sname
CREATE-MT $sname NAMES STUDENTS M T sname
```

In the above example, the first statement creates the abstract object type STUDENTS. The second creates the descriptor object type NAMES, whose members are variable-length strings. The third statement creates a map type called sname from STUDENTS to NAMES and specifies that it is a single valued, total map type. And the fourth statement creates the inverse of sname, a multi-valued, total map type.

As an example of retrieval, consider a query to retrieve the names of the students who got a grade of 3.5 or better in the course "CS101". The following processing scheme could be used to answer the above query.

1. Store in the set G, all GRADES objects that are greater than 3.5;
2. Store in the set E1, all destination objects of the access path G : $grade, i.e. all ENROLLMENTS objects whose grades are greater than 3.0.
3. Store in the set E2, all destination objects of the path {"CS101"} : $c# o $course, i.e. all ENROLLMENTS objects whose course o c# is "CS101".
4. Let $E = E1 \cap E2$;
5. Store in the set RESULT, all destination objects of the access path E : belongsto o name; i.e names of the students owning the ENROLLMENTS objects.

Notice that the same result could be obtained by a different query which isolates STUDENTS objects instead of ENROLLMENTS objects. This method may be less efficient because of the fact that the subpath comprising the map "belongsto" will be more heavily traversed than in the first method.

Data transfer operations are provided for transferring values from the storage structures to the data structures in the host language. These operations pla a significant role in providing data independence since they can perform necessary type conversion. For example, a report writer at the conceptual level may view a data element as a string even though that data element may be stored as an integer. The transformation from integer type to string type is done at the intermediate level. If the type of this data element is later changed to real type, this change is transparent to the said report writer. The notion of *cursor* is used for input/output. Logically, the cursor points to the object to be read next. A cursor is associated with each variable of the types AOS or DOS, or access path. Initially, the cursor is positioned at the start of the first object. The intermediate transaction (in actual syntax using host programming language C) for the query above is given below:

```
main()
{
  char *name;
## DEFINE-DOS  G  GRADES /* declare G to hold a subset of GRADES */
## DEFINE-DOS  N  NAMES  /* N to hold a subset NAMES */
/* declare E1, E2, and E3 to hold subsets of ENROLLMENTS */
## DEFINE-AOS  E1,E2,E3  ENROLLMENTS
/* get into G all grade values of 3.5 or above in the type GRADES */
## GET-DOS G GRADES [3.5,*)
/* define X, an access path from GRADES to ENROLLMENTS */
## DEFINE-ACC-PATH  X  G:$g
/* define Y, an access path from CNOS to ENROLLMENTS */
## DEFINE-ACC-PATH  Y  "CS101".$cno.takenby
## GET-OS E1 X   /* get ENROLLMENTS objects for grade >= 3.5 */
## GET-OS E2 Y   /* get ENROLLMENTS objects for cno = "CS101" */
## INTSCT-OS E3 E1 E2 /* E3 = E1 ∩ E2 */
## DEFINE-ACC-PATH  Z  E3.belongsto.name
## GET-OS  N  Z
## GET-NEXT-OBJ name N cs
## while (!(STATUS[4])) {
     printf("%s\n",name)
##   GET-NEXT-OBJ name N cs
   }
}
```

Another example of the intermediate transaction is shown in section 5.

4. The Physical Model

In this section, the term *normalized file* will be used to refer to a file whose records have fixed number of fields, and each field stores a single value. Normalized files are used as physical constructs in most of the existing database systems because such structures have one-to-one correspondence with the constructs of many conceptual models, and because the storage space management for normalized files is less complicated than that associated with other file structures. But the normalized files have serious limitations both in representing the semantics of the real world information [20, 21, 22], and enhancing the performance of a database system. From the performance point of view, the limitations stem from the fact that records of a normalized file can only represent linear relationships of individual data values. Linear relationships amongst groups of data values, or non linear relationships, i.e., tree-structured relationships, have to be decomposed into linear relationships before they can be represented by normalized files. Such decomposition is detrimental to the performance of a database system. Specifically, the drawbacks are:

1. If an attribute of an object has more than one value (i.e., multi-valued attribute), then the attribute has to be stored in a secondary record, thus requiring extra time to access the attribute values.

2. Access paths cannot be represented in its general form. For example, an access path from UNITS to STUDENTS in the schema shown in Fig.3-1 could not be represented as the structure would have to capture the nesting of STUDENTS within ENROLLMENTS within COURSES within UNIT.

Thus, to enhance the performance of a database system, a richer construct is needed for the physical level of a database system. Another point to consider in choosing the physical constructs for evolvable physical databases is that they be restructurable, and such restructuring should preserve facts, i.e., it should neither delete nor insert facts from/into a database. Furthermore, for restructuring to be efficient, the keys of the records should be as short as possible. Record segmentation, an example of restructuring, is most economical if records have short keys, which are repeated in each segment. Thus records whose first fields are keys, and which can have repeating groups as remaining fields are an ideal form of storage structure. Such a record structure is called a *List* in the physical data model used in ADAMS. Another reason for introducing the List-based structure is to be able to represent *access tree*; two or more access paths emanating from an object. For example, if the access paths S:sname and S:enrolledin o course o cno are accessed together frequently for a student S, then performance could be improved by storing the access tree containing these two access paths. An access tree having n length-one paths, each having the destination set of cardinality one is similar to a normalized record having n fields.

Non-linear structures have been studied by researchers both as conceptual tools and physical storage structures. The structures are commonly referred to as non first normal relations [1, 18, 26, 32], or compacted relations [5, 6]. The design of the List-based storage structure is influenced by the work of Abiteboul [1]. The List-based storage structure and its associated operations that form the physical model are described below.

4.1. The List-Based Storage Structure

The modeling construct of the List-based storage structure is a List given by the following recursive definition.

Definition: A List is an n-tuple, $(k,m_1,...,m_{n-1})$, $n > 0$, where k is called the *List-Id*, and each m_i is a set of Lists and is called a *molecule*.

A List-Id is either the symbol $\perp$ (which indicates a null value); or an object drawn from some *domain*, e.g., a number drawn from a domain named INT, a character string drawn from a domain named STRINGV.

No two Lists which are elements of the same molecule have the same List-Id.

The valid domain names are: OBJ-REF, LIST-REF, P-INTn, INTn REALm:n, F-STRINGn, V-STRING, and DATE. Except for the first two domains, they have the same meaning as the domains of the intermediate model. The domain OBJ-REF contains a set of values that are used to represent abstract objects. LIST-REF is a domain of relative addresses that are used to point to other Lists.

An example List instance is given in figure 4-1. Here, molecules are underlined with the symbol "=", and Lists are underlined with the symbol "-". Intuitively, the example

```
<f1,{<s1,{<c1,{<3>},{<B>}>,<c2,{<4>},{<A>}>}>}>
               ---   ---        ---   ---
              ===== =====      ===== =====
          ---------------- ----------------
         ===================================
     ----------------------------------------
    ==========================================
-----------------------------------------------
```

Figure 4-1: An Example of List

could be considered as a file (List-id f1 being the file name) of one record instance with key s1. The record instance has one multi-valued field (molecule), the values of this field are records each of which has a has a key c_i, and two fields, each containing a single value. The example may be interpreted as a List containing information about a student. The List represents the facts about the courses the student is taking, and for each course, the number of units and the grade he/she receives. A molecule X is called a *sub-molecule* of a molecule Y if X is inside Y. For example, the molecule instance {⟨B⟩} is a sub-molecule of the molecule instance {⟨c1,{⟨3⟩},{⟨B⟩}⟩,⟨c2,{⟨4⟩},{⟨A⟩}⟩}.

The structure of a List resembles that of a tree, with the List-Id being the root of the tree. The reason that it is called a List rather than a tree lies in the semantics of the data values in the nodes of the tree. There are two kinds of data values: atomic values, and addresses. Addresses point to other Lists that may be shared by different nodes, or may point to predecessor nodes. Thus, logically the structure is a List. If the addresses are treated as just atomic values, a List may be viewed as a tree. Therefore, in subsequent discussion a List will be considered, and portrayed as a tree. The domain name from which the List-Ids of the Lists in a molecule are drawn, will be called the *root domain* of the molecule. For example, if the List of figure 4-1 belongs to a molecule M, and if object f1 is drawn from domain F, then the root domain of M is F. Since all the Lists of a molecule have the same structure, the shape of these Lists may be described by the *format* of the molecule. The format of a molecule carries the following information:

1. the number of Lists in it, i.e., the cardinality of the molecule,

2. the domain name of the List-Ids, and

3. the formats of the molecules in the Lists, if there are any.

The cardinality of a molecule can be one of four values: zero or one, exactly one, more than zero; and more than one. The format of a molecule can be graphically represented as a tree having four types of nodes. Each node is the root of some subtree that represents a sub-molecule. A node type indicates the cardinality of the molecule and a node label (written inside the node) shows the domain name. Figure 4-2 shows an example of a format where a triangular shaped node represents the fact that the sub-molecule for which it is the root domain has the cardinality of zero or one. Similarly, rectangular, pentagonal, and hexagonal shaped nodes indicate cardinalities of exactly one; zero or more; and one or more respectively. Intuitively, the format says that the molecule has zero or more instances of List whose List-Ids are drawn from domain A, and each of which has two molecules, say M1 and M2. M1 has one or more instances of List whose List-Ids are drawn from domain B and each List instance has two molecules, etc.. In linear syntax, a parenthesized expression is used to specify the format of a molecule. Each pair of parentheses is followed by one of the symbols '▪',''','*','+' to denote cardinalities of zero or one; exactly one; zero or more; and one or more respectively. Thus the format corresponding to figure 4-2 would be (S(C(U)'(G)▪)+)* .

Formally, a format is defined as a string in the language generated by the following grammar. Let t be the name of a domain.

$$F \rightarrow t\ R$$
$$R \rightarrow (\ F\ )^{\blacksquare}\ R \mid (\ F\ )'\ R \mid (\ F\ )^{*}\ R \mid (\ F\ )+\ R \mid \epsilon$$

Some example formats are OBJ-REF (OBJ-REF (REAL)▪)* and OBJ-REF (STRINGV)' (OBJ-REF(REAL)▪)*. The format of a molecule describes the cardinality of the relationships that a List in a molecule represents. A format of the form X(Y)▪ means that the List-Id is drawn from domain X, and that either one List of molecule Y or the null-value symbol $\perp$ is in the molecule of the List. This means that an object of domain X is related to at most one tuple of set Y. A format of the form X(Y)' means that each object of domain X is related to exactly one tuple of Y, a format of the form X(Y)* means that each object of domain X is related to zero or more tuples of Y, and a format of the form X(Y)+ means that each object of domain X is related to

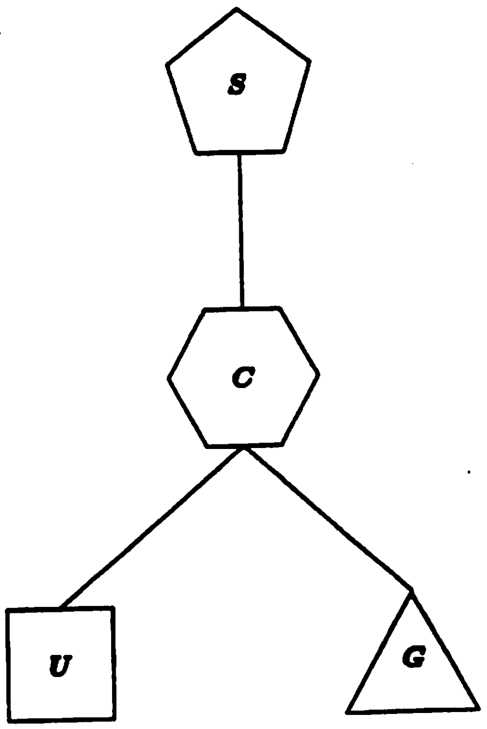

Figure 4-2: Example Format

one or more tuples of Y. A molecule described by the X(Y)ⁿ is used to represent a single-valued partial map type of an intermediate schema. The formats X(Y)', X(Y)*, and X(Y)+ are used for single-valued total, multi-valued partial, and multi-valued total map types respectively.

Formally, the families $\mathcal{M}(f)$ of molecules associated with formats f, are defined on the structure of f as follows. Let dom(t) be a set of objects belonging to domain t, and $\wp(S)$, for some set S denote the power set of S.

- if f = t for some domain type t, then $\mathcal{M}(f) = \wp(\text{dom}(t))$
- if $f = t\ X_1 ... X_n$ then $\mathcal{M}(f)$ is the collection of finite subsets A of $\{\text{dom}(t) \times P_1(X_1) \times ... \times P_n(X_n)\} \cup \{<\perp, P'_1(X_1), ..., P'_n(X_n)>\}$ where
  1. no two elements of A have the same first coordinate,

2. if $X_i = (f_i)^{\bullet}$ then $P_i(X_i) = m; m \in \mathcal{M}(f_i) \wedge |m| \leq 1$,

3. if $X_i = (f_i)'$ then $P_i(X_i) = m; m \in \mathcal{M}(f_i) \wedge |m| = 1$,

4. if $X_i = (f_i)^*$ then $P_i(X_i) = m; m \in \mathcal{M}(f_i) \wedge |m| \geq 0$,

5. if $X_i = (f_i)+$ then $P_i(X_i) = m; m \in \mathcal{M}(f_i) \wedge |m| > 0$, and

6. $P_i'(X_i) = m; m \in \mathcal{M}(f_i) \wedge |m| > 0$,

such that if a tuple $<x,Y_1,...,Y_n>$ occurs as an element of some set, where each Y_i is a set of tuples, and if Y_i, $1 \leq i \leq n$, contains a tuple $<\perp,Z_1,...,Z_m>$, where each Z_j is a set of tuples, then no tuples in Z_j, $1 \leq j \leq m$, must have the symbol $\perp$ as the first component.

Intuitively, this last restriction ensures that no two objects would be related via a null value. For example, $\mathcal{M}$(OBJ-REF)$^{\bullet}$ is a set of abstract objects, M(OBJ-REF (STRINGV)')$^{\bullet}$ is a set of List instances each of which has an abstract object as List-Id and a variable length string as a sole molecule. This set could also contain a List instance of the form $\langle\perp,\{\langle\text{string1}\rangle,...,\langle\text{stringn}\rangle\}\rangle$.

In the List-based storage structure, information is stored as List-Ids. Because of the recursive nature of the structure, accessing information from the structure involves recursively descending the tree. In order to give an intuitive example, assume that a pointer P is pointing to a List L of a certain molecule M. Suppose L has 2 molecules that are numbered 1 and 2. These numbers are called the *Relative Molecule Number* RMN and are given in left-to-right order of occurrence of molecules with respect to a root node. A List with id x in any one of these two molecules can be accessed by letting a pointer P' point to the List. This is done by the operation LocateList (explained later) which takes a pointer to a List (P in this case), an RMN value (say 1 in this case), and a List-Id (x in this case), and returns a pointer to the List with List-Id x in the RMNth molecule of L. We have assumed that initially some pointer P is pointing to some List. In the actual implementation this is accomplished by giving names to molecules, and using the operation OpenMolecule to initialize the pointer. Thus the addressing scheme used to access data items of a List-based storage structure can be viewed as having two kinds of addresses: *absolute address* and *relative address*. When a molecule is created, it can be given a name for future reference. This name is the absolute address of the molecule. A molecule within a List instance can be addressed by a relative addressing scheme in which the address has two parts *P* and *n*, where *P* is pointer to the beginning of the List instance, and *n* is RMN. For example, if P were pointing to the List instance ⟨a,{⟨b1,{⟨c1⟩},{⟨d1⟩}⟩,⟨b2,{⟨c1⟩},{⟨d2⟩}⟩},{⟨e1⟩}⟩, then (P,1) would refer to the molecule {⟨b1,{⟨c1⟩},{⟨d1⟩}⟩,⟨b2,{⟨c1⟩},{⟨d2⟩}⟩}, and (P,2) would refer to the molecule {⟨e1⟩}. As a short-hand notation, we will use P↑ to refer to the List instance pointed to by P. The operations on the List-based storage structure are given in the following section. The operations are provided as functions, and the relative address of a molecule, if needed, is supplied as two parameters: pointer variable name, and RMN.

4.2. Operations on the List-Based Storage Structure

Operations are provided to create and destroy molecules; create, locate, insert, and delete List instances; and reorganize molecules. These operations are provided as functions written in C, that can be called from a program. If there is no error the functions return the value 0. An error code is returned and the operation is not executed, if the execution of a function would violate the constraints imposed on a List by its format. Inside a program, variables are used to point to a List instance. The following operations are for creation and manipulation of the constructs of the List-based storage structure.

CreateMolecule (*Mname*, *Format*):

> This function is used to create a molecule. *Mname* is the name given to the molecule and must be unique within a database. *Format* is the format of the Lists of the molecule.

OpenMolecule (*ptr*, *Mname*):

> *Mname* is the name of a molecule. This function creates a List instance with the name as List-Id, and the molecule *Mname* as a sole molecule. Then *ptr* is made to point to the List instance.

LocateList (*iptr*, *optr*, *RMN*,*id*):

> This operation is used to locate a List instance in a given molecule. The molecule is identified by a relative address consisting of a List pointer *optr* (outer List pointer), and an RMN. *iptr* (inner List pointer) is the name of a pointer to point to the beginning of a List inside the molecule. *id* is a List-Id of the list to be located.

LocateFirst (*iptr*, *optr*, *RMN*):
Locates the first List instance in the given molecule.

LocateNext (*nptr*, *optr*, *RMN*):
Locates the List instance immediately following the last read List instance in the given molecule.

ReadId (*ptr*, *var*): Reads the List-Id of *ptr*↑ into *var*.

AddList (*iptr*, *optr*, *RMN*):
iptr↑ is inserted into the molecule if the List-Id does not already exist. If the List-Id already exists, or if the List instance to be inserted is not in proper format then the List instance is not inserted and an error code is returned.

SubtractList (*List-Id*, *optr*, *RMN*):
The List instance with the given List-Id, if it exists, is removed from the molecule, if it would not violate the constraints imposed on the molecule.

CreateList (*ptr*, *id*, *nmol*):
Creates a List instance with id as List-Id, and nmol number of molecules, all of them empty. Ptr is set to point to the newly created instance.

As an illustration of the use of these operations, consider the following C program which retrieves the grade for each course that student s1 is taking from the List given in 4-1. Assume that f1 is the name of the outermost molecule.

```
main() {
  ListPtr p,p1,p2,p3;
  OpenMolecule(p,"f1");
  LocateList(p1,p,1,s1);
  LocateFirst(p2,p1,1);
  do {
    ReadId(p2,crs);
    LocateFirst(p3,p2,2);
    ReadId(p3,grd);
    printf("%s %s\n",crs,grd);
  } while(LocateNext(p2,p1,1) == 0);
}
```

Reorganization of the List-based storage structure is done by the two operations:

merge, and *split*. The merge operation combines two molecules X and Y producing a third molecule Z. It is analogous to outer join as proposed by Codd [10]. The submolecule of X into which the molecule Y is to be merged is specified by a preorder molecule number PMN, which is the number of the root node of a submolecule in the preorder enumeration of the format tree of the molecule. For example if X is specified by the format A(B)•(C(D)*)• and Y is specified by the format C(E)*, then the operation •merge X Y 3 Z• would create Z of the format A(B)•(C(E)*(D)*)•. Here, 3 is an PMN specifying the molecule (C(D)*)•. The resultant molecule Z will contain the union of all the facts stored in X and Y.

The syntax of merge operation is given below:

MERGE *M1 M2 PMN M3*:

> *M1* and *M2* are names of the two Molecules to be merged. Let f1 and f2 be their formats. *PMN* is the preorder molecule number in M1 on which to merge the two molecules. The result is stored in a molecule named *M3* whose format f3 is obtained by replacing the domain at position *PMN* in f1 with f2.

A molecule X can be split into two molecules Y and Z by the split operation. The submolecule to be split is specified by an PMN, and an RMN. The RMN specifies the relative molecule number within the List defined by PMN. For example, if the format of X is A(B)•(C(D(E(F)•(G)•)•)•(H)•)• and the operation •split X 3 1 Y Z• is executed, then the format of Y will be A(B)•(C(H)•)• and that of Z will be C(D(E(F)•(G)•)•)•. Intuitively, if the main List is viewed as a tree, the PMN specifies the node of the tree and the RMN specifies the branch emanating from that node which is to be split.

The syntax of the SPLIT operation is given below.

SPLIT *M1 PMN RMN M2 M3*:

> where M1 is the name of the molecule to be split. PMN and RMN specify the submolecule to be extracted. The Lists in the submolecules are extracted from the Lists of M1 and stored in M2. The remaining submolecules are stored in M3.

The algorithms to implement these operations are given in [2]

4.3. An Implementation Technique

The operations on the List-based storage structure have been implemented on Unix[6] and PCDOS[7] operating systems using the programming language C. In this version, each named molecule is represented as a file (a sequence of bytes). A List instance as illustrated in the preceding sections contains redundant symbols and is not an efficient way to represent it sequentially. Therefore, parenthesized expression is used to store a List in a file. In this method, the symbols "(", ")", and "," are used to separate data items. For example, the List of example 4-1 will be represented as f1(s1(c1(3)(B),c2(4)(A))).

The sequence of bytes that form a file are divided into two kinds: *control bytes*, and *data bytes*. A control byte is distinguished from a data byte by having its most significant bit set; that is, bit 1 of control byte is always 1. This means that only seven bits of each data byte can be used for storing data. This poses no problem for storing ASCII strings but storage of numbers in binary form could cause problems as bit 1 may be set in a data byte. Thus all numbers are stored as a sequence of base-128 digits with the type SHORT-INT occupying 1 byte, INT occupying 2 bytes, and LONG-INT occupying 4 bytes. Real numbers are stored in 8 bytes with the first four bytes representing the number and the second four bytes representing the fraction.

Control bytes are used as separators and each control byte contains three fields:

Field	Bits	Purpose
flag	1	always set for control byte
symbol	2 to 4	1 for "," , 2 for "(", 4 for ")", 5 for "),", 6 for ")("
count	5 to 8	the number of ")"s

If the format of a molecule contains a plus symbol or an asterisk symbol, meaning multiple occurrences of List instances, then the List instances are separated by commas. As an example, consider the format A(B)'(C(D))*. Some example List instances are:

[6]Unix is a trademark of Bell Laboratories.

[7]PCDOS is a trademark of International Business Machines.

a1(b1)(c1(d1),c2(d2)),a2(b2)(c3(d3)),...

where each li, where l is a letter and i is a digit, is assumed to be drawn from the domain L. If we represent the control byte whose symbol code is s and whose count is c as ⟨s:c⟩, then the above instance would be stored as follows:

a1⟨2:0⟩b1⟨6:1⟩c1⟨2:0⟩d1⟨5,1⟩c2⟨2,0⟩d2⟨5,2⟩a2⟨2,0⟩b2⟨6,1⟩c3⟨2,0⟩d3⟨5,2⟩...

To save space some of the control bytes are removed if such removal would not cause ambiguity in interpretation of the List. For example, assume that in the above example representation of objects of the domain A, B, and D require fixed number of bytes, then the List could be represented in a compact form as follows:

a1b1⟨6,1⟩c1d1c2d2⟨5,2⟩a2b2⟨6,1⟩e3C+⟨5,2⟩...

A named molecule is organized as a hash file where the hash keys are the List-Ids of the constituent Lists. To locate a List with a given List-Id in a given molecule, first the hash function associated with the molecule is applied to the List-Id. This yields a home block number which either contains the List or has a pointer to the overflow area where the List is stored. If a home block is completely filled with pointers, and an attempt is made to insert a new List instance into the block, then the whole file is reorganized by assigning twice the previous number of blocks and using a new hash function.

The variable used to point to a List actually has two pointers: a data pointer that points to the beginning of a List, and a format pointer that points to the format of the molecule that contains the List. The format is organized as a tree whose nodes can be described by the following data structure (shown in the syntax of programming language C):

```
struct Node{
  int FCBSkip, FDBSkip, NCBSkip, NDBSkip;
  char FCByte, NCByte;
  struct Node *sibling, *child;
}
```

To locate the first List of a molecule M, identified as P.n for some pointer P, the sibling pointer must be followed n-1 times starting from the node pointed to by the format pointer part of P. This will locate a node in the format tree which contains information about the data pointer movement. Data pointer is first moved till the control byte given in FCByte is sensed FCBSkip number of times. Then the data pointer is moved FDBSkip number of bytes to reach the first occurrence of the Lists in M. Similarly, the NCByte, NCBSkip, and NDBSkip are used to locate the next occurrence of the Lists in M. For example, Suppose M is a molecule whose format f is A(B(C)*)*(D(E)*)*, where instances of domains A, B, C, and D are of fixed length occupying 2, 3, 4, and 2 bytes respectively, and instances of domain E are of variable length. Consider an instance of M shown in figure 4-3. Here, x... for a lowercase letter x denotes a value drawn from domain X, and the length of the value is the number of dots plus 1. x- means a value drawn from X but the length is variable. A symbol enclosed within angle brackets is to be interpreted as a single byte whose value is given in hexadecimal code within the brackets. When M is opened (using OpenMolecule operation), the data pointer of P, denoted P.d, points to the first List of M, and the format pointer, denoted P.f, points to the root of the format tree. If at this time an operation to locate the molecule addressed as P.2 is issued, then first P.f is adjusted to become identical to the sibling pointer of P.f, and is shown as P.f' in the diagram. The node pointed to by P.f' shows how P.d should be moved to locate the first List of the said molecule. In the example, the information contained in the node says that the data pointer must be moved until the first occurrence of the control byte having the value hexadecimal E2. The byte immediately following is the start of the first List of P.2. In the diagram this is shown by the pointer P.d'.

A List is created by CreateList operation. This operation allocates enough space in the main memory that can be used to build a more complicated List recursively using AddList operations. When a main-memory-resident List is to be inserted into a molecule, the List is checked against the format of the molecule to ensure that it satisfies the constraints imposed by the format.

5. Binding

Binding is the process of translating structures, constraints, and operations of one DBE, called the *source environment* to another, called the *target environment*. It is a function provided by a database management system. There are two kinds of bindings in ADAMS: conceptual-to-intermediate, and intermediate-to-physical binding. This section

```
a.b..c...c...<D1>b..c...<E2>d.e-<90>e-<D1>d.e-<D2>
-                           -
|                           |
P.d                         P.d'


P.f --->   0        ---- P.f'--->   E2   FCByte
           0       |                 1   FCBSkip
           0       |                 0   FDBSkip
           D2      |                D1   NCByte
           1       |                 1   NCBSkip
           0       |                 0   NDBSkip
           .   ----                  /   sibling
   ---     .                ---      .   child
  |                        |
  |                        |
   -->     0                -->      0
           0                         0
           2                         3
           D1                       90
           1                         1
           0                         0
           /                         /
   ---     .                         /
  |
  |
   -->     0
           0
           3
           0
           0
           4
           /
           /
```

Figure 4-3: An Instance of a Molecule and its Format Tree

describes the conceptual-to-intermediate binding briefly and intermediate-to-physical binding in detail.

5.1. Binding a Conceptual DBE to an Intermediate DBE

The complexity of the algorithm to bind the structures, constraints, and operations of a conceptual database environment to an intermediate database environment depends on the data model used for the conceptual database. The simplest case is the absence of a conceptual model; i.e., the intermediate model is used as a conceptual model. However, because of the primitive and procedural nature of the operations of the intermediate model, this may not be intuitive to the end-user. Another approach would be to use the modeling constructs of the intermediate model along with a high-level non-procedural manipulation language such as FQL [7], as a conceptual model. A third possibility would be to use the functional data model DAPLEX [34] at the conceptual level. If this were the case, then for each n-argument function $f: D_1 \times \ldots \times D_n => R$ of DAPLEX, a new object type D' must be introduced in the intermediate model. Then a function from each D_i to D' must be calculated. Finally, a function from D' to R has to be built. The binding of operations is also straight forward. If the conceptual model is the relational data model then the binding of structures and operations becomes more complicated.

In translating the transactions of a source environment, knowledge about the cost associated with different realizations at the target environment must be considered, and the one that has the least cost must be chosen. Since the intermediate level does not possess any device dependent characteristics, the cost measure cannot be very realistic. A good rule is to minimize the estimated total number of map types that must be used when a transaction of the target environment is evaluated. One way to choose an efficient target transaction is to make sure that a map type is never traversed more than once. This is an area that must be further investigated.

5.2. Binding an Intermediate DBE to a Physical DBE

The binding of an intermediate DBE to a physical DBE is divided into two components: binding of structures, and binding of operations. Since the constraints of an intermediate database are on the structures only, their binding is tied to the binding of the structures.

5.2.1. Binding of Structures

An abstract object of an intermediate database is represented at the corresponding physical database by a natural number. The numbers are unique within a generalization hierarchy only. This means that a number n represents one abstract object in one generalization hierarchy and a different abstract object in another generalization hierarchy. A list of intervals of numbers that are currently free, i.e., not representing any object, is maintained for each generalization hierarchy, and the smallest number of the list is used each time an object is created. A set of objects is thus a set of natural numbers and can be represented as a bit map. If the bit maps are sparse they may be compressed and stored. The performance advantage of the use of bit maps as compared to a list of numbers, and a compression technique are given in [3]. Each type/subtype is thus represented as a bit map and the question of whether an object, represented by a number n, belongs to a type/subtype can be answered by inspecting whether the bit n is on in the bit map of that type/subtype. The list of free numbers for each generalization hierarchy is the bit map for the root type of the generalization hierarchy, and the number to be assigned to a newly created object of the hierarchy is just the number corresponding to the first zero from the start of the bit map.

A descriptor object is represented by its value. Unless frequent searches are made on a descriptor object type, it is not explicitly maintained. If searches are made on a descriptor type, the the values in the type will be arranged as a B+ tree [12] if the values are integers; as a digital tree (trie) [24], if the values are character strings; and as an interval hierarchy [40] if the values are real numbers.

A map type of an intermediate database may be, in general, represented in one or more named molecules of the corresponding physical database. When a map type M from object type $\mathcal{X}$ to object type $\mathcal{Y}$ is defined at the intermediate level for the first time, the system will generate instructions to create a molecule. This molecule will have the format X(Y)$^{\bullet}$ if M is single-valued partial, X(Y)' if M is single-valued total, X(Y)* if M is multi-valued partial, and X(Y)+ if M is multi-valued total map type. The molecule will be given a system-generated name, say S (in actuality, S is an integer number). The fact that map type M is represented in molecule S is recorded in the dictionary for the intermediate-to-physical binding. At this time a map type has a one-to-one relationship with a molecule, and mapping in such manner is called the canonical mapping. The one-to-one relationship between the map types and the molecules may not hold once the physical design process is initiated. As will be explained in the following chapter, the physical database designer may merge molecules to enhance performance. Furthermore, these mergers may leave the original molecule intact, i.e., a duplicate copy of the molecule may be merged with another molecule. Map types represented in a molecule form a tree structure, and only those map types whose source object types are the root of the tree are accessible from outside. For example, suppose the map types M_1: $\mathcal{U} => \mathcal{V}$, M_2: $\mathcal{U} => \mathcal{W}$, and M_3: $\mathcal{V} => \mathcal{X}$ are represented in a molecule S having the format $\mathcal{U}(\mathcal{V}(\mathcal{X})^*)^*(\mathcal{W})^*$, then the map types M_1 and M_2 are accessible from outside because their source object type $\mathcal{U}$ is the root of S. Map type M_3 is not accessible from outside but is accessible only through map type M_1. In subsequent discussion, the map types that can be accessed from outside will be called ***externally accessible map types*** or EAMT for short and the set of map types of some molecule M that are externally accessible will be denoted as EAMT(M).

Since a map type may be stored in more than one molecule, the problem of finding the appropriate molecule to be used in accessing information stored in a map type, must be addressed. A procedure explained later in this section binds access paths to molecules.

5.2.2. Binding of Transactions

An intermediate transaction contains data manipulation operations of the intermediate data model that has to be pre-processed into a program in pure host language containing appropriate declarations and calls to run-time routines. The run-time routines are divided into the following categories:

1. routines to record the usage information and the statistics,

2. routines to determine the storage structures to be used,

3. routines to minimize the cost of accessing the storage structures,

4. routines to realize the intermediate level operations.

An intermediate transaction manipulates data by means of access paths. The pre-processor scans the transaction to determine the access paths that are referenced together. This is done by keeping track of the access paths having the same source object. For example, consider the following transaction to print the course and grade of a student with a given name that is entered from a terminal.

```
main()
{
  ## DEFINE-AOV e  ENROLLMENTS;
  ## DEFINE-AOSV  EN  ENROLLMENTS;
  ## DEFINE-DOSV  CN  CNOS;
  ## DEFINE-DOSV  GR  GRADES;
  char name[30], cno[6];
  float agrade;
  scanf("%s",name);
  /* Isolate ENROLLMENTS objects for the name */
  ## DEFINE-ACC-PATH ENR name:$sname.enrolledin
  ## GET-OS  EN  ENR
  /* for each enrollment object do */
  ## GET-NEXT-OBJ  e  EN  as;
  while (!(STATUS[4])) {
   /* get the course number */
   ## DEFINE-ACC-PATH CRS e:course.cno;
   ## GET-OS  CN  CRS
   ## GET-NEXT-OBJ cno  CN  cs;
   /* get the grade */
   ## DEFINE-ACC-PATH GRD e:grade;
   ## GET-OS  GR  GRD;
   ## GET-NEXT-OBJ *agrade GR  re;
   printf("%s  %f\n",cno,grade);
  }
}
```

The pre-processor first scans the transaction and gathers the map type names used in the transaction and sorts the access paths into groups. In each group are the access paths that have the same root object. The algorithm to determine whether two access paths X and Y belong to the same group works as follows. If the root objects of X and Y are both constants, and they are identical, then X and Y are in the same group. If the roots of X and Y are the variables r_x and r_y respectively, then X and Y are in the same group if r_x is the same as r_y, and the variable is not changed between two access path definitions. For example, the three access paths defined in the above program will be divided into two groups. The first group will have the access path "name:$sname.enrolledin", and the second group will contain the access paths "e:course.cno" and "e:grade". Notice that the two access paths in the second group have the same variable as the roots and that the variable is not changed between the definitions. The access paths in a group are compactly represented as an *access tree*. The access tree for this example would be ENROLLMENTS:(course.cno,grade). The root of an access tree is the object type from which the roots of the constituent access paths are drawn. Thus in the above example the root of the access path is ENROLLMENTS, since e is drawn from that type. Notice, that as the name implies, access trees are strictly trees. For example, the two paths X:a.b.e and X:c.d.e would form the access tree X:(a.b.e,c.d.e) as opposed to the directed acyclic graph like X:(a.b, c.d).e . The reason for this is that although the leaf types of the two access paths are the same, in general they are different object sets. Since an access path is a special case of an access tree, in subsequent discussion, access paths will be referred to as access trees.

After the first pass of the transaction, the pre-processor generates a series of structure variable declarations in the host language. A structure created for each map type consists of the name of the map type which is initialized, and the numbers of inserts, deletes, and updates performed on the map type. A structure created for each access path consists of the access path specifications, and the cardinality of the root. A structure created for each object type consists of the name of the object type, number of objects inserted, the number of objects deleted, and if it is a descriptor type, the average length of the objects inserted. A constant denoting transaction number is also declared. The value of the constant is obtained from a persistent variable whose value shows the number of the last transaction. The pre-processor increments the value and uses it to define the constant. Then the pre-processor scans the transaction again to generate code. The DEFINE statements are translated into appropriate declarations in the host language. The variables to hold object sets are declared as a host variable of type pointer to character string. This character string is used to store a number of objects separated by an appropriate character. After the variable declarations have been generated, the pre-processor generates a statement for each access tree that it has constructed. This statement is a call to the run-time routine of the second category, that searches for the optimal set of molecules to be used to realize the access tree. The heuristic search strategy used to find the set of molecules is given in section 5.2.3. The routine constructs a process which accept the root object set of the access tree, and the desired access path name as input, and returns the leaf object set. The compiled process is stored for the transaction, so that subsequent execution of the transaction may use this code if dependent storage structures have not been changed.

After the code for each insert, delete, or update operation involving a map type, the pre-processor generates code to increase the count in the corresponding structure. After the code to realize each object set that is the root of an access tree, the pre-processor generates code to update the cardinality of the root of the access tree. When the transaction has been processed, for each of the structures described above the pre-processor will generate a call to the run-time routine that will write out information recorded in that structure to the appropriate location on the disk.

The functions in the third category are called before each abstract object set is used in an access path. These functions arrange the objects in the set into ascending order of their hash values so that the maps that are in the same block are placed together. This is to avoid reading the same block twice. Run-time routines in the fourth category implement the intermediate level operations.

5.2.3. Procedure to Find the Optimal Set of Molecules

The sole run time routine in the second group is responsible for finding the minimum number of molecules that should be used to realize an access tree[8] We first explain the terms needed to describe the algorithm. If T is an access tree with root R, the map types having R as the source type are said to be at level one. The map types whose source type is the same as the destination type of a map type at level n is said to be at level n+1. The largest level number for any map type in a tree T is the height of T, and is denoted as HEIGHT(T). For a map type m in an access tree T, the *successors* of m in

[8]The greedy heuristic does not always find the minimum number of molecules; the problem of finding minimum number of molecule is NP-complete. However, if the molecules are designed using the methodology described in section 6, it does give minimum number of molecules for frequently used access trees.

T, denoted SUCC(m,T), is a set of map types that emanate from the destination node of m. For example, in the access tree T of figure 5-1, map types a,b, and c are at level one, and the remaining map types are at level two. HEIGHT(T) is 2, and SUCC(a,T) is the set {d,e,f}. A heuristic procedure that is used to find the optimal set works under the assumption that a map type M participates as an EAMT in exactly one molecule. This assumption is consistent with the physical design methodology used in ADAMS. Under this assumption, if there are n map types emanating from an object type, then there are up to n molecules containing these map types as EAMTs. The procedure given below finds a minimum set of molecules needed to realize an access tree.

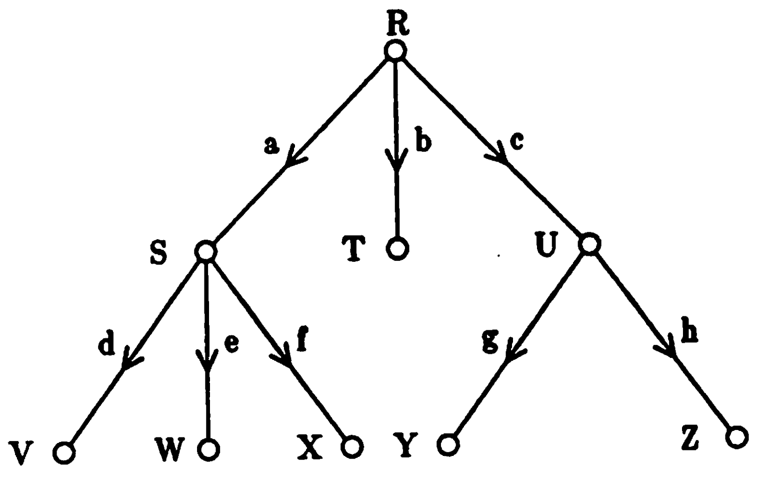

Figure 5-1: An Example Access Tree

```
procedure Find (T: AccessTree);
begin
   /* Initially all the map types are unmarked */
   i := 1;
   while (i <= HEIGHT(T)) do
   begin
      for each unmarked map type x of T at level i do
      begin
        find molecule M such that $x \in$ EAMT(M)
        R := R $\cup$ {M};
        mark(x,M)
      end;
      i := i + 1
   end
end
procedure Mark(p: MapType, M: Molecule);
begin
```

```
  mark map type p of T;
  for each map type x in SUCC(p,T)
    if x ∈ SUCC(p,M) then mark(x,M)
end
```

The set of molecules to be used to realize T is returned in the set R. When an intermediate level transaction is executed for the first time, the above heuristic procedure is used to determine the molecules that should be used to realize the access trees in the transactions. The bindings of these access trees are recorded along with the molecules used, so that subsequent executions of the transactions may use the same bindings if none of the dependent molecules has been changed. This approach is similar to the binding method used in System R [4].

6. The Physical Design Methodology

The canonical mapping of map types of an intermediate DBE to molecules of a physical DBE is usually not efficient due to either or both of the following reasons.

1. If a transaction requests for n attributes (values of n maps), then n molecules have to be accessed. Thus the performance of such transaction may be inferior to the case where all the attributes are stored in a record; like in most conventional database systems.

2. If a transaction uses an access path of length greater than one, then the corresponding molecules have to be composed. Composition is analogous to relational join and is costly.

Performance deterioration due to the first reason can be remedied by clustering the map types that emanate from an object type so that the probability of finding all n maps requested by a transaction in one molecule is high. This problem is similar to the attribute partitioning problem [14, 27, 17, 16, 28, 31]. The approach taken in ADAMS to overcome the poor performance caused by the second reason is to merge frequently composed molecules and maintain it as a separate molecule. This is possible because the storage structure allows the nested information to be represented.

Thus the physical design methodology of ADAMS can be epitomized as a decision procedure to merge molecules to improve the performance of a database system. Molecules are merged in two ways: *clustering*, and *composition*. Clustering is the merging of two or more named molecules on their roots. The roots must be of the same type. For example two molecules with the formats X(Y)* and X(Z)* could be clustered to get a molecule with the format X(Y)*(Z)*. Composition is the merging of two molecules based on an non root type of the first molecule with the root of the second molecule. For example, the two molecules described by the formats X(Y)* and Y(Z)* could be composed to get a molecule described by the format X(Y(Z)*)*. The physical design process consists of two phases. In the first phase, each object type of the intermediate schema is considered in turn. The map types emanating from this object type are grouped into non-overlapping clusters based on the usage pattern. This phase is thus referred to as *Clustering Phase.* In the second phase, map types that should be composed are considered. The second phase is termed *Composition Phase.* Each of these phases is explained in detail in the following subsections with the help of the usage pattern given in figure 6-1.

Int. transaction	Freq	Root Size	Access Tree	Type
E = var.$n.e;	100	1	N.$n.e	R
while(getnext(x,E))			E.(c.cno,g)	R
printf("%s %d\n",				
x.c.cno,atoi(x.g));				
print(var.$cno.u)	10	1	CN.$cno.u	R
getde(X,G,"[3.5,*)");	20	6	G.$g	R
E1 = X.$g;		1	CN.$cno.t	R
E2 = var.$cno.t		20	E.b.n	R
E3 = E1 ∩ E2;				
printf(E3.$e.n);				
E1 = var.$n.e;	200	1	N.$n.e	R
E2 = var.$cno.t		1	CN.$cno.t	R
E3 = E1 ∩ E2;		1	E.g	U
update(g,(E3,*),(E3,3.0))				

Figure 6-1: An Example Usage Pattern

6.1. The Clustering Phase

The objective of this phase is to clump together the map types emanating from each object type into a number of molecules so that map types that are referenced together most often reside in the same molecule. Clustering is done by first assuming that all the map types emanating from an object type are grouped into one cluster. Based on some measure, this cluster may be split into two clusters, in which case each of the two clusters are considered for further splitting. The process goes on until further splitting is not beneficial. The core algorithms of this phase are the Bond Energy Algorithm BEA [30], and the attribute splitting algorithm [31].

In the first step of the clustering phase, the usage pattern is scanned, and for each object type of the intermediate database, a *map type usage matrix* MTU, and a *frequency vector* FV are set up. An MTU shows the map types that are referenced by each transaction, and an FV shows the frequency of occurrence of the corresponding transaction. The rows of MTU correspond to transaction numbers and the columns correspond to the map types. The MTU and the FV for the object type ENROLLMENTS using the usage pattern of figure 6-1 is shown in figure 6-2.

	belongsto	course	grade	frequency
t1	0	1	1	100
t2	0	0	0	10
t3	1	0	0	20
t4	0	0	1	200
		MTU		**FV**

Figure 6-2: Map Type Usage Matrix and Frequency Vector

In the second step, the MTU and FV are used to build a *map type affinity matrix* MTA, for each object type. The matrix MTA is a symmetric square matrix which records the affinity among the map types i and j as a single number MTA(i,j) [= MTA(j,i)]. The affinity measure MTA(i,j) is the sum of the frequencies of all transactions that use both map types i and j. The MTA derived from the above MTU and FV is shown in figure 6-3.

In the third step the BEA is used to transform MTA into a *Clustered Map Type*

$$\begin{array}{l|ccc} & \text{belongsto} & \text{course} & \text{grade} \\ \text{belongsto} & 20 & 0 & 0 \\ \text{course} & 0 & 100 & 100 \\ \text{grade} & 0 & 100 & 300 \end{array}$$

MTA

Figure 6-3: Map Type Affinity Matrix for ENROLLMENTS

Affinity Matrix CMTA. The BEA is a heuristic procedure for permuting rows and columns of a square matrix in order to group numerically larger array elements together. The algorithm is typically applied to partition a set of interacting variables into subsets which interact minimally. The example map type affinity matrix above is one of the solutions produced by the BEA.

In the fourth step, CMTA matrix is used to group the map types emanating from an object type into non-overlapping clusters. This is done by first splitting a CMTA matrix X into two matrices X1 and X2 if such splitting is possible (based on some objective function to be explained later). The two CMTA matrices X1 and X2 are then considered for splitting. This process goes on until no further splitting is beneficial. Map types in each of the resultant matrices are grouped into one cluster.

The binary splitting algorithm on CMTA matrix X for a certain object type T is described using the following notations. Let M(T) be a set of all map types emanating from the object type T. Let S(X) be a set of map types participating in X, and $\bar{S}$ be M(T) - S(X). Let T(X) be defined as follows:

$$T(X) = \{t \mid (\sum_{x \in S(X)} MTU(t,x)) \geq 2 \wedge (\sum_{x \in \bar{S}(X)} MTU(t,x)) = 0\}$$

Intuitively, T(X) is set of transactions that use two or more map types of a matrix X and use no other map types emanating from the same object type. Thus it is a set of transactions that can be processed efficiently if map types in X are put in one partition.

Also let CT(X) be defined as follows:

$$CT(X) = \sum_{k \in X} FV[k]$$

Intuitively, CT(X) is the total number of times the transactions access map types in the set X.

A given n $\times$ n CMTA matrix X can be split into two non-overlapping matrices XU and XL by an algorithm called SPLIT2. The algorithm works as follows. A point p between two diagonal elements of X is first considered. The point p partitions X into two CMTA matrices XU, the upper left part of X, and XL the lower right part of X. The cost of this partition scheme is computed using the following formula:

$$z = CT(T(XU)) \times CT(T(XL)) - (CT(T(X) - (T(XU) \cup T(XL))))^2$$

If the cost is positive then the partition is feasible. By placing p at n - 1 different locations in n different "forms" of X (each form obtained by moving the leftmost column to the extreme right and the top row to the bottom) up to n(n-1) feasible partitions can be generated. The partition with the greatest z value is then taking as the solution. If there is no feasible partition, then the matrix X is not considered for further clustering. The partitions for the example CMTA is:

partition 1:	{course, grade}
partition 2:	{belongsto}

Figure 6-4: Partitions for ENROLLMENTS

For illustrative purpose, we will use the following convention to name molecules for the example database. Each molecule representing a single map type will be given the name that is the first three letters of the map type's name (the dollar symbol prefix is not counted). If two molecules whose names are a and b, are merged on the root then the name of the resultant molecule is "a+b", and if they are merged on some domain that is the destination set of a map type, say x, then a period followed by the second molecule's name is appended to the molecule's name corresponding to x. For example if the molecule whose name is "a+b" is merged with the molecule named x, on the second domain of the first molecule, i.e., the destination set of a, then the resultant molecule will be named "a.x+b". The above partition generates a molecule whose name is cou+gra .

The algorithm to partition a set of map types emanating from an object type T can now be stated using the SPLIT2 algorithm explained above. Assume that SPLIT2 returns true if a feasible partition exists.

```
procedure SPLIT(X);
if SPLIT2(X,Y,Z) /* if X can be split into Y and Z partitions */
then begin
      SPLIT(Y);
      SPLIT(Z)
   end
else output X;
```

Each partition could be realized by merging the map types in the partition on the List-Id. Appropriate merge instructions are generated after the map types have been partitioned. For the example usage pattern, the clustering phase of the physical design would produce only one extra molecule; the one that results from merging the two molecules corresponding to course and cno map types.

6.2. Composition Phase

Whereas the clustering phase considers merging map types emanating from a single object type, the composition phase considers merging map types emanating from different object types that can be used to form access paths. In terms of the physical model, this means merging two molecules based on the object type other than the List-Id of the first molecule. The idea is to improve performance by storing a frequently-composed set of map types in a single molecule.

In the first step, a multigraph G = (V,E), called the *composition graph* is set up. The set of nodes V is the union of two sets T and C, where T is the set of object types of an intermediate schema, and C is a set of molecules obtained from the clustering phase. Each node of C is assigned a number indicating the estimated number of blocks incurred for insert, delete, and update operations on the molecule. Initially, E is empty. For example, if the object type names of the example database are abbreviated by the first two letters of the names, the set T would be {ST, ID, NA, EN, CO, GR, CN, UN}, and the set C would be {sno, $sno, sna, $sna, bel, enr, tak, cou+gra, $gra, cno, $cno, uni, $uni}.

Each arc of G denotes a composition of molecules at the end of the arc and is given two kinds of labels: the P label denotes the PMN number of the source molecule on which the molecules are composed, and the W label denotes the total cost of compositions. Thus E may be considered as a set of 4-tuples <m,n,p,w>, where m and n are molecules, p is the PMN number on which they are composed and w is the cost of the composition. Among the arcs connecting any two molecules, no two of them will have the same P label. For an arc e, let P(e) and W(e) denote the P label and the W label respectively. In order to describe how the arcs are constructed, we need a procedure similar to procedure Find of 5.2.3. This procedure, which we will call Find2, uses the same algorithm to find the subset of a given set of molecules to be used for an access tree, but instead of generating code, it returns a tree data structure. Each node of the tree carries the name of the molecule and a set of pairs of values. The first value of each pair is a pointer to the molecule to be composed with, and the second value is the PMN number of the domain on which to compose. The parameters for Find2 are: the set of molecules, a pointer to an access tree, and a pointer to point to the root of the tree data structure. As a second step in the composition phase, the arcs of G are constructed using the following procedure.

```
1. For each transaction t in the usage pattern do the following steps.

2. For each access tree a of t do:

   a. Find2(V,a,ptr);

   b. Compute the cost w of composing the root R of a with the first
      molecule M1 of ptr↑;

   c. If an arc exists from R to M1 in G
      then add w to the weight of the arc
      else create arc and assign weight w.

   d. For each composition of molecules m and n on PMN p in ptr↑ do:

      i. Compute w the estimated number of blocks incurred due to
         composition of m and n on p.

      ii. If there is an arc from m to n whose P label is p
          then add w to the W label of the arc
          else construct an arc e from m to n and
               assign P(e) = p, W(e) = w.
```

The cost of processing transactions in the usage pattern, C, is computed by adding the weights of all the arcs and the labels of the nodes. The value of C is the estimated total number of block accesses for the transactions in a given usage pattern. For the example database, the composition graph is shown in figure 6-5.

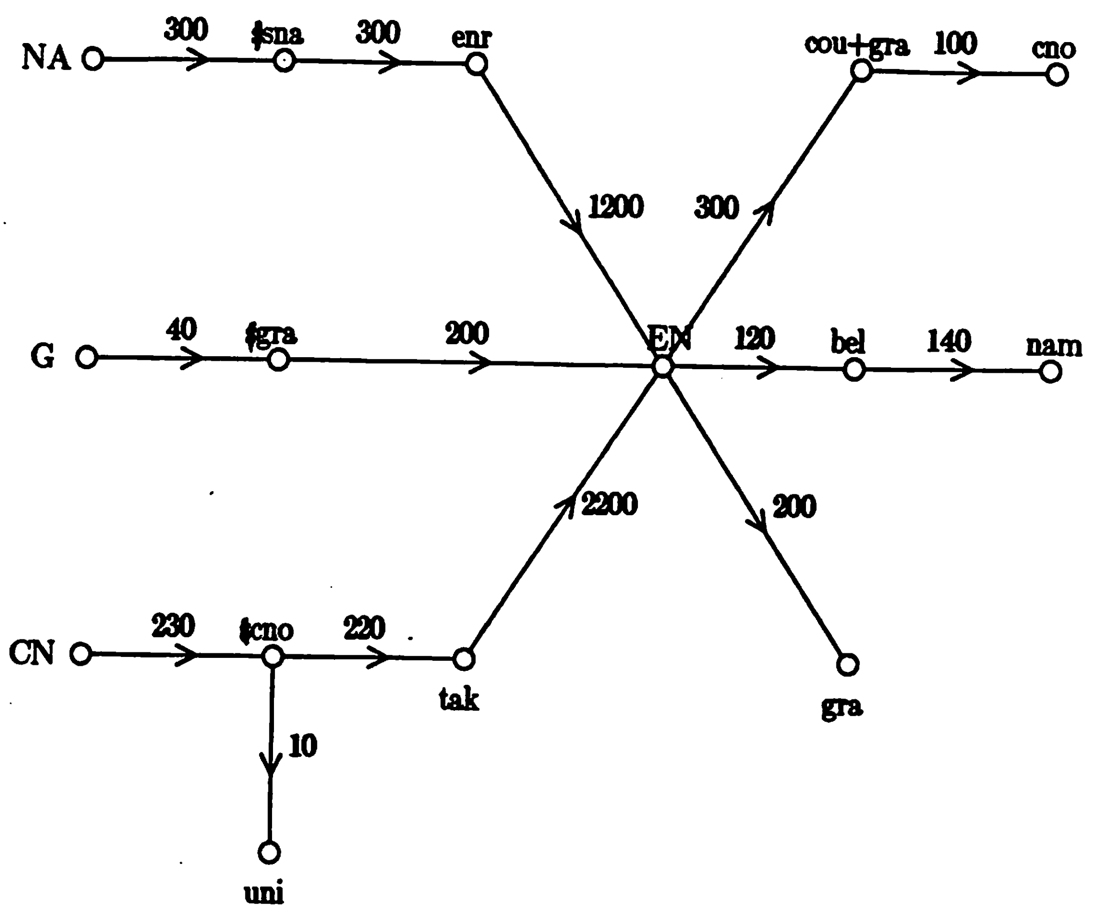

Figure 6-5: The Graph G for the Running Example

The procedure to decide which molecules should be merged uses the following two functions:

EXTERN(M) where M is a molecule, returns the externally accessible molecules of M. For example if M were described as a.x + b.y + c then EXTERN(M) would be a + b + c.

COST(Ptr,m,n,p) where Ptr is a pointer to the data structure returned by Find2, m and n are molecule names, and p is a PMN number. This function returns the total cost composing molecules m and n on PMN p in the scheme given by Ptr↑.

Intuitively, the procedure works as follows. Let T be the total space for the data, C be the cost of processing the transactions using the molecules of G, and S be the total space required by molecules of G. Initially all the arcs of G connecting a node in T to a node in C are marked; the rest are unmarked. If G contains an arc which is unmarked and the total space used by the molecules in V is less than the available space then the arc with the largest label, say the arc e connecting molecules m and n, is considered for merging. Two tentative schemes of mergers are considered: one in which m and n are merged leaving n intact, and the other in which m and n are merged but n is replaced with EXTERN(n). In both cases the number of block accesses due to insert, delete, and update operations on the resultant molecule is computed. Then the access trees affected by the merger are inspected to compute changes in weight of the arcs whose end points contain m or n or both. Then the total cost and storage space for each of the tentative schemes are computed. If both schemes are feasible, then the one giving the bigger cost savings per storage unit is taken as the current solution. The arc e is marked and the process repeated. The procedure is described by the following algorithm.

```
1. While ($\exists$ a $\in$ E such that a is unmarked and $W(a) > 0$) and ($S < T$) do steps
   2 to 10.

2. Let e = $<M1,M2,p_e,w_e>$ be a tuple in E such that $\forall$ x $\in$ E $W(e) \geq W(x)$.

3. Compute the parameters for M3, the resultant molecule if M1 and M2 were
   to be merged on PMN $p_e$.

4. Let $X_0$ be the set $\{<m,n,p,w> \mid <m,n,p,w> \in E$ and $m = M1$ or $m = M2$
   or $n = M1$ or $n = M2\}$

5. Construct the sets $V_1$ and $V_2$ as follows:

   a. $V_1 = V - \{M1\} \cup \{M3\}$;

   b. $V_2 = V_1 - \{M2\} \cup M2'$ where $M2' = EXTERN(M2)$;

6. Construct the sets $X_1$ and $X_2$ as follows:
   For each access tree t that uses M1 and/or M2 of V do

   a. Find2(V1,t,ptr).

   b. For each composition of molecule m with molecule n on some PMN p in
      ptr$\uparrow$
```

```
        if m = M3 or m = M2 or n = M3 or n = M2
        then let $\delta$ = COST(ptr,m,n,p).

        if <m,n,p,w> for some w is in $X_1$
        then increase w by $\delta$
        else $X_1 = X_1 \cup$ <m,n,p,$\delta$>.

    c. Find2(V2,t,ptr).

    d. For each composition of molecule m with molecule n on some PMN p in
       ptr↑

        if m = M3 or m = M2' or n = M3 or n = M2'
        then let $\delta$ = COST(ptr,m,n,p)

        if <m,n,p,w> for some w is in $X_2$
        then increase w by $\delta$
        else $X_2 = X_2 \cup$ <m,n,p,$\delta$>.

 7. Compute $c_0$, $s_0$, $c_1$ , $s_1$ , $c_2$ , $s_2$ as follows:
    a. $c_i = \sum_{x \in X_i} W(x)\ 0 \leq i \leq 2$
    b. $s_i = \sum_{x \in X_i} S(x)\ 0 \leq i \leq 2$
 8. if $c_1 < c_0$ and $S - s_0 + s_1 < T$ then $r_1 = (c_0 - c_1) / (s_1 - s_0)$ else $r_1 = 0$

 9. if $c_2 < c_0$ and $S - s_0 + s_2 < T$ then $r_2 = (c_0 - c_2) / (s_2 - s_0)$ else $r_2 = 0$

10. if $r_1 = 0$ and $r_2 = 0$ then mark e of V
    else if $r_1 \geq r_2$ then
    begin
      $V = V_1$; $E = E - X \cup X_1$;
      $C = C - c_0 + c_1$;
      $S = S - s_0 + s_1$
      generate instruction to merge M1 and M2 on $p_e$ giving M3
    end
    else begin
      $V = V_2$; $E = E - X \cup X_2$;
      $C = C - c_0 + c_2$
      $S = S - s_0 + s_2$
      generate instruction to merge M1 and M2 on $p_e$ giving M3
      generate instruction to realize EXTERN(M2)
```

```
end
```

The above algorithm stops when either G has no arc or all the arcs have been marked. Since in each iteration an arc is either eliminated or marked, the algorithm will eventually terminate. The run time of the algorithm is the product of the three quantities ϵ, α and λ, where ϵ is the number of edges of G, α is the number of access trees in the usage pattern, and λ is the maximum number of compositions of molecules in an access tree. To reorganize the List-based structures, first the existing molecules are split so that one molecule represents one map type of the intermediate DBE. Then the new molecular configuration is derived by interpreting the instructions generated from the two phases of the physical design process. A more efficient reorganization scheme is currently being investigated.

7. Cost Estimation

In this section we describe the formulae for estimating the cost of composing molecules, and the cost of storing pre-composed molecules. We make an assumption that each insertion, deletion, or update operation on a map type at the intermediate level involves one map. The following information, used in the cost formulae, is stored for each molecule. For a molecule M_j:

1. the average length L_j of Lists in M_j,
2. the average number of Lists N_j in M_j,
3. the storage space S_j, in blocks, used by the molecule,
4. For each submolecule m specified by PMN n, of M_j:
    a. q^j_n the average cardinality of m,
    b. p^j_n the average number of Lists of M_j in which a List of m occurs.
5. I_j, the estimated number of block accesses due to insertions into M_j,
6. D_j, the estimated number of block accesses due to deletions from M_j,
7. U_j, the estimated number of blocks due to updates on M_j.

If M_j represents a single map type, the p_2^j and q_2^j correspond to the cardinality of the relationship which is supplied by the user; I_j,D_j, and U_j correspond to the insert, delete, and update operations performed on the map type (since each operation involves a single map, it incurs one block access), and the N_j and S_j values are stored as statistical data, and L_j can be computed form the lengths of the object involved and the values of p_2^j and q_2^j If M_j is a molecule resulting from merging two molecules M_1 and M_2, then the parameters of M_j can easily be computed from the parameters of M_1 and M_2. For a newly-created molecule M_j, S_j is assigned the value $1.25{*}N_j/(\lfloor B/L_j \rfloor)$, where B is the block size. This is the number of home buckets at 80% packing density for a hash file. As overflow blocks get created the value of S_j will change.

Consider an access path R o T_1 o ...o T_x, where each T_i is a map type. Suppose that it corresponds to the molecular composition R: o M_1 o ... o M_y, $1 \leq y \leq x$, with each M_i containing N_i Lists of average length L_i. Let the number of distinct objects of the leaf type of the access path R : M_1 o ... o M_{j-1}, $j \leq y$, be k. Since these k objects are arranged into ascending order of their hash values by a run time routine, the expected number E_j of blocks accessed to form the access path R : M_1 o ... o M_{j-1} o M_j is:

$$E_j = S_j \cdot [\, 1 - \prod_{i=1}^{k} (N_j d - i + 1)/(N_j - i + 1)]$$

where

$d = 1 - 1/m$, and

$k \leq N_j - \lfloor N_j/S_j \rfloor$

The above formula was developed by Yao [41]. In the actual implementation, the closed form approximation of the formula [39] is used.

The number of distinct objects in a leaf type of an access path is computed using a novel formula that takes the p and q values into account. Detailed derivation of this formula and the measurement of its accuracy are shown in appendix 1. The number of distinct objects k' in the leaf type of the access path R : M_1 o ... o M_{j-1} o M_j, the last two molecules are composed on PMN n, is estimated as follows:

$$k' = N_j(1 - \prod_{i=1}^{kg_n^{j-1}} (1 - p_n^j/(np_n^j-(i-1))))$$

8. Performance Comparison

To illustratively compare ADAMS with a relational database system, two extreme cases are considered. In one extreme (case 1), the user specifies no indexes, and in the other extreme (case 2), he/she specifies indexes on all attributes of all relations. This section presents the results of simulation of performance of ADAMS as compared to both cases in the target environment using the relational model. The example database of figure 3-1, and the example usage pattern of figure 6-1 were used as test cases. Access cost is measured in terms of number of blocks read. The join algorithm for case 1 of the relational database is Inner/Outer Loop method, and the join algorithm for case 2 is Join-Index method. All files are assumed to be organized as hash files with 100% packing density and no records in overflow areas. Furthermore, we assume that a set of records can be arranged according to their hash keys and thus no block will be read more than once in accessing the set of records. The results are summarized below:

Approach	Case	Space	Time
Relational	1	33	5670
Relational	2	56	1790
ADAMS	no phy. des.	55	2180
ADAMS	after phy. des.	59	1660

Here, after the physical design phase, the physical database of ADAMS entails 7.26% less block accesses over the fully inverted relational system; the space increment is 5.35%. Of course, this example usage pattern is not typical of the target environment since access paths are short. When the access path x o $unit o takenby o belongsto o name was included in the usage pattern the following results were obtained:

Approach	Case	Space	Time
Relational	2	56	5390
ADAMS	after phy. des.	62	3060

Here a 43.22% reduction in number of block accesses is obtained at the expense of

9.67% increment in space.

9. Conclusions

This paper has presented an approach to allow the physical storage organization of a database system adapt to the prevailing usage pattern with minimum human intervention. Unlike previous work in adaptive database systems, the approach presented herein views a conceptual database as a set of logical access paths of arbitrary lengths connecting database objects. Transactions against a database are characterized in terms of the logical access paths they reference. These access path reference patterns are monitored while the database is in use, and are stored as access trees. At some user-determined time, this pattern is used to redesign the physical storage configuration.

The approach taken is to directly represent the most frequently used logical access (sub)tree at the physical storage level, thereby reducing the time required to construct them at run time. The physical design methodology has two phases, the first of which is similar to the conventional attribute partitioning, the second can be termed as composition. A physical storage structure called the List-based storage structure is used to represent trees. The storage structure provides restructuring operations that facilitate incremental physical (re)design. An experimental prototype system (ADAMS), that incorporates the above ideas has been implemented. Performance studies show very encouraging results.

References

[1] Abiteboul, S. and Bidoit, N.
Non First Normal Form Relations to Represent Hierarchically Organized Data.
In *Proc. Symp. on Principles of Database Systems.* ACM SIGACT-SIGMOD, Waterloo, Canada, April, 1984.

[2] Ahad, R.
A List-Based Storage Structure for Personal Databases.
In *Proc. of the Symp. on Small Systems.* ACM, Danvers, Mass., May, 1985.

[3] Ahad, R.
User-Assisted Design and Evolution of Physical Databases.
PhD thesis, Dept. of Computer Science, Univ. of Southern Calif., April, 1985.

[4] Astrahan, M. M., et al.
System R: Relational Approach to Database Management.
ACM Transactions on Database Systems 1(2), June, 1976.

[5] Bancilhon, F. et. al.
Verso: A Relational Backend Database Machine.
In *Proc. Inter. Workshop on Database Machines.* San Diego, 1982.

[6] Bancilhon, F., Richard, P., and Scholl, M.
On Line Processing of Compacted Relations.
In *Proc. Intl.Conf. on VLDB*, pages 263-269. VLDB, Mexico, September, 1982.

[7] Buneman, P. and Frankel, R.
FQL--A Functional Query Language.
In Bernstein, P (editor), *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, pages 52-58. ACM-SIGMOD, Boston, May, 1979.

[8] Chan, A.
A Methodology for Automating the Physical Design of Integrated Databases.
PhD thesis, MIT, September, 1980.

[9] Codd, E.F.
A Relational Model for Data for Large Shared Data Banks.
CACM 13(6):377-387, June, 1970.

[10] Codd, E.F.
Extending the Database Relational Model to Capture More Meaning.
ACM Transactions on Database Systems 4(4), December, 1979.

[11] Codd, E.F.
Relational Database: A Practical Foundation for Productivity.
CACM 25(2):109-117, February, 1982.
The 1981 ACM Turing Award Lecture.

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

[13] Dearnley, P.
A Model of a Self-organizing Data Management System.
Computer Journal 17(1):13-16, 1974.

[14] Eisner.
Mathematical Techniques for Efficient Record Segmentation in Large Shared Databases.
ACM Journal 23(4), October, 1976.

[15] Hammer, M. and Chan, A.
Index Selection in a Self-Adaptive Database Management Systems.
In *Proc. ACM SIGMOD*, pages 1-8. ACM, 1976.

[16] Hammer,M. and Niamir,B.
A Heuristic Approach to Attribute Partitioning.
In *Proc. Int. Conf. on Management of Data.* ACM-SIGMOD, Boston, 1979.

[17] Hoffer, J. A. and Severance, D. G.
The Use of Cluster Analysis in Physical Database Design.
In *Proc. First Int. Conf. on Very Large Databases.* IEEE, 1975.

[18] Jaeschke, G. and Scheck, H.J.
Remarks on the Algebra of Non First Normal Form Relations.
In *Proc. ACM SIGACT-SIGMOD Conf..* ACM, Los Angeles, 1982.

[19] Kamel, N and King, R.
PACKRAT: A Self-Adaptive Support System for Multiple Queries.
Technical Report, Computer Science Dept., Univ. of Colorado, Boulder, 1985.

[20] Kent, W.
Data and Reality .
North Holland, 1978.

[21] Kent, W.
Limitations of Record-Based Information Models.
ACM TODS 4(1), 1979.

[22] Kent, W.
A Realistic Look at Data.
Database Engineering 7(4):22-27, December, 1984.

[23] Knuth, Donald E.
The Art of Computer Programming.
Addison-Wesley, 1970.

[24] Knuth, Donald E.
The Art of Computer Programming.
Addison-Wesley, 1973.

[25] Lyngbaek, P. and McLeod, D.
A Personal Data Manager.
In *Proc. of the Tenth Int. Conf. on VLDB.* VLDB, Singapore, August, 1984.

[26] Makinouchi, A.
A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model.
In *Proc. Inter. Conf. on VLDB.* IEEE, Tokyo, 1977.

[27] March, S. T. and Severance, D. G.
The Determination of Efficient Record Segmentations and Blocking Factors for Shared Data Files.
ACM transactions on Database Systems 2(3), September, 1977.

[28] March, S. and Scudder, D.
On the Selection of Efficient Record Segmentations and Backup Strategies for Large Shared Databases.
ACM Transactions on Database Systems 9(3):409-438, September, 1984.

[29] Martin, J.
Computer Data-Base Organization.
Prentice-Hall, Englewood Cliffs, N.J., 1975.

[30] McCormick, W.T., Schweitzer, P.J., and White, T.W.
Problem Decomposition and Data Reorganization by a Clustering Technique.
Operations Research 20(4):741-751, July, 1972.

[31] Navathe, S. et.al.
Vertical Partitioning Algorithms for Database Design.
ACM Transactions on Database Systems 9(4):680-710, December, 1984.

[32] Scheck, H.J. and Pistor, P.
Data Structures for an Integrated Database Management and Information Retrieval System.
In *Proc. Int. Conf. on VLDB.* IEEE, Mexico, 1982.

[33] Schkolnick, Mario.
A Survey of Physical Database Design Methodology and Techniques.
In *Proc. of the Fourth Int. Conf. on Very Large Databases*, pages 474-487. IEEE, 1978.

[34] Shipman, D.W.
The Functional Data Model and the Data Language DAPLEX.
ACM Transactions on Database Systems 6(1):140-173, March, 1981.

[35] Stocker, P.M. and Dearnley, P.A.
Self-organizing Data Management Systems.
Computer Journal 16(2):100-105, 1973.

[36] Stocker, P.M. and Dearnly, P.A.
A Self-organizing Data Base Management System.
In Klimbie, J.W. and Koffeman, K.L. (editor), *Data Base Management*, pages 337-349. IFIP, Corsica, France, April, 1974.

[37] Stonebraker, M., Wong, E. Kreps, P., and Held, G.
The Design and Implementation of Ingres.
ACM TODS 1(3):189-222, Sept., 1976.

[38] Tsichritzis, D. C. and Klug, A.
The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Ssytems.
Information Systems 3, 1978.

[39] Whang, K., Wiederhold, G.,and Sagalowicz, D.
Estimating Block Accesses in Database Organization: A Closed Noniterative formula.
Comm. of ACM 26(11):940-944, November, 1983.

[40] Wong, K.C. and Edelberg, M.
Interval Hierarchy and Their Application to Predicate Files.
ACM Transactions on Database Systems 2(3):223-232, Sept, 1977.

[41] Yao, S. B.
An Attribute Based Model for Database Access Cost Analysis.
ACM TODS 2(1):45-67, 1977.

[42] Yormark, B.
The ANSI/SPARC DBMS Model.
North Holland, Amsterdam, 1976.
The ANSI/X3/SPARC/SGDBMS Architecture.