1986

An extended relational data base management system for engineering design

Charles Fung

Rochester Institute of Technology
School of Computer Science and Technology

An Extended Relational Data Base Management System for Engineering Design

by

Charles Fung

August 6, 1986

A thesis, submitted to the Faculty
of the School of Computer Science and Technology,
in partial fulfillment of the requirements for
the degree of Master of Science in Computer Science

Approved By:

1-23-87
Professor Jeffrey Lasky (Chairman)

1-26-87
Professor Chris Comte

-23-Jan-87
Professor Andrew Kitchen

I, Charles M. Fung, hereby grant permission to the Wallace Memorial Library, of Rochester Institute of Technology, to reproduce my thesis in whole or in part. Any reproduction will not be used for commercial use or profit.

Feb 19, 1987

ABSTRACT

At present, the use of the relational model in the engineering design support domain is restricted due to the following: lack of ability to handle complex objects, no support for Abstract Data Types, inappropriate concurrency control for long transactions, no support for versioning and update propagation, poor efficiency, and insufficient design rule checking and consistency contraints enforcement. A simple relational database management system is designed and implemented under the UNIX• operating system to incorporate two major extensions: support of user-defined Abstract Data Types and operators, and built-in Design Data Versioning. The design, implementation, and possible extensions to these new facilities are described.

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

ACKNOWLEDGMENTS

The author would like to express gratitude to Professor Jeffrey Lasky, the chairperson of the thesis committee, and to the other members of the committee for their invaluable guidance and help.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Area of Interest

The focus of this thesis is the use of relational data base management systems in engineering environments. Specifically, the thesis considers two extensions to the relational model which appear attractive for engineering design environments. These extensions are the support of user-defined abstract data types and operators, and design data versioning.

Digital computers are increasingly being used to automate different aspects of engineering design and manufacturing. Typically, computer support for these activities has been implemented as a collection of disjoint programs, and so the sharing of data between applications has usually involved resource-intensive data conversion. While data base management systems (DBMS) have been well developed and widely adopted in the business community as a powerful data processing tool, their application in the engineering environments has only recently gained attention.

From a user's perspective, integration of different phases of the design process implies the capability to perform complex operations, such as design capture, analysis, or synthesis, on design information at any level of abstraction without concern for the underlying mechanism of data storage. As engineering designs become more complex, the costs of such integration through manual data conversion soon become prohibitive, due to high (typically) maintenance costs. It has become apparent that a centralized data base, containing all of the design and manufacturing data, would be the most effective and efficient approach for supporting comprehensive design automation.

1.2. Application of the Relational Model to Engineering Design

Compared to other data models such as hierarchical and network models, the *relational* model provides a more flexible *view* of the underlying data. To the engineer who may not have much knowledge about data structures, the data is presented in its most natural form, which is tables (or more strictly speaking, relations). The navigable links between the relations are transparent and easily comprehensible. Most important of all, the *relational* model and its corresponding *relational algebra* permit relationships to be dynamically created. Relationships are implicitly updated whenever the value of an item changes. The data schema can be easily altered without the need for major changes to application programs and the data base. The real power of the *relational* model comes from its *relational processing capability* provided by its algebra, using operations such as *select*, *project* and *join*. The ability of the user to obtain different *views* of the data through the use of the *relational algebra* facilitates the efficient sharing of data between applications as well as the dynamic changing of the data schema.

Furthermore, in the engineering design environments, the data and the schema of the data base are usually in a dynamic state. Because of the high degree of data independence maintained in the relational data base, these fluctuations are efficiently accommodated. Therefore, most researchers agree that the *relational* system is the natural candidate for engineering design applications.

Currently available relational data base management systems exhibit major inadequacies when their suitability to engineering applications are explored. The research results reported here include a design and implementation of a relational data base management system that incorporates several enhancements to the relational model which are targeted at overcoming some of these deficiencies.

1.3. Problem Identification

Early attempts in introducing DBMS to various engineering fields were made by adopting existing systems, already in use, to store and manage engineering data. The reason for using "off the shelf" DBMS was, of course, to save the time and money otherwise needed to design a totally new system. Unfortunately, most of those attempts were unsuccessful, because commercially available DBMS were almost invariably designed specifically for business use. Many of the earlier attempts, as presented in [MASA74], [KORE75] and [WONG79], were evaluated and found slow and awkward to use. They were expected to generate solutions to all the problems of engineering design automation, but all were found lacking.

A comparison of the characteristics of business and engineering data bases presented in [SIDL80] reveals a number of discrepancies between what commercial DBMS offer and what an engineering data base requires.

In general, the following are the major areas of identified deficiencies in existing relational data bases.

- Lack of ability to handle complex objects
- No support for abstract data types
- Inappropriate concurrency control for long transactions
- No support for versioning and update propagation
- Poor efficiency
- Insufficient design rule checking and consistency contraints enforcement

1.4. Background

1.4.1. Overview

Numerous attempts have been made to solve some or all of the problems related to the use of relational data bases in the engineering design domain. Although most of this research was carried out separately, it was generally aimed at common targets. Although some attempts failed, quite a few did grow to maturity with at least partial success. More noteworthy examples include the *Mosaic Object Manager* [ATWO85], IBM *System R* [LORI83], IBM *SQL/DS* [LORI85], the experiences of [STON83] and [BARA85] in *INGRES*, *CORD Data System* [VU85], and Boeing *IPIP* [JOHN85].

Instead of tracing each research attempt from its time of conception to date, a grouping of different research efforts under specific major areas of interest is presented. This will be followed by a presentation of some interesting products which are commercially available.

1.4.2. Abstract Data Types and Complex Objects

Design support applications are some of the most complex artifacts of human endeavor in the sense that they attempt to support one of the activities of the human mind most difficult to reduce to rule and number -- creation.

Design activities confronted by engineers today are getting so complex that some design goals are impossible to achieve without the help of computer aided design tools. This situation is most prominent in the dynamic world of Very Large Scale Integrated (VLSI) Circuit design. As pointed out in [GADI85], the VLSI design process can be, in an already simplified manner, decomposed into six hierarchical levels: a system level, an Instruction Set Architecture (ISA) level, a Register Transfer Level (RTL), a logic level, a device level and an artwork level. System level design aids, such as the Architecture Design and Assessment System (ADAS), includes design capture, synthesis and analysis or verification tools that operate at the system, ISA, or RTL levels. In comparison, chip level design aids, such as KIC from

University of California, Berkeley, which uses the CalTech Intermediate Form (CIF), perform the same generic functions but operate at the logic, device, or artwork level.

One way through which people normally deal with complexity is abstraction, and one of the problems of introducing data base technology into design automation is that most design objects require attributes with more general data types than those supported in current DBMS.

To this end, researchers in DBMS have turned to the use of *Abstract Data Types* (hereafter referred to as ADTs), which already have been extensively investigated in programming language contexts. Early exploration in this area can be found in [LOCK79], [ROWE79], [SCHM78], and [WASS79].

As defined in [STON83], an ADT is an encapsulation of a data structure, with its implementation detail hidden from the user, along with a collection of related operations defined on this encapsulated structure. In order to use ADTs in a data base, their existence has to be registered in the data base and then the allowable operations have to be defined by procedures written in a language that supports both data base access and the ADTs. Using ADT's, the designer can define and manipulate complex design objects at an abstract level, without worrying about their low level details. ADTs can also limit the access to a relation in prespecified ways, thereby guaranteeing a higher level of data security and data integrity.

[LOOM85] introduced a three-architecture framework for achieving a step toward integration through logical data modeling. The three architectures are *Information*, *Computer Systems* and *Control Architectures*, which can be directly mapped into the *three-schema* approach at the logical data level, namely the *External*, *Internal* and *Conceptual Schemas*. The *three-schema* approach has been adopted by the IGES (International Graphics Exchange Standard) Committee in their development of the architecture for PDES (Product Definition and Exchange Standard). In their logical data model, an *entity* is defined to be a class of an object (physical or abstract) and its properties are termed its *attributes*. There are three fun-

damental types of relations between entities: *Generalization*, *Association* and *Aggregation*. A *Generalization* relationship enables a set of similar entities to be viewed as a single entity (in artificial intelligence terminology, this is an "*a-kind-of*" or "*is-a*" relationship). An *Association* relationship relates entities that are different but not necessarily on the same level of abstraction (in artificial intelligence terminology, this is an "*an-instance-of*" relationship). An *Aggregation* relationship enables a collection of dissimilar entities to be viewed as one entity. (In artificial intelligence terminology, this is a "*consists-of*" or "*a-part-of*" relationship).

While *Association* relationships have always been supported by all commercial DBMS, the rest are new data types. *Aggregation* relationships are better known to the computer science research community as *complex objects*; whereas *generalization* relationships basically encapsulate most of the work being done on what has been broadly termed *Abstract Data Types*. Apart from more detailed descriptions of the properties of these relationships, the authors in [ATWO85] also introduced their *Mosaic Object Manager* which formed a testbed for their data model. *Long fields* (an instance of ADT) contain arbitrarily long bit strings and character strings. For this ADT, unusual recovery and buffering techniques are employed and are discussed in [BANE85] in relation to systems implementation. The proposed solution is to maintain a pool of different sized blocks to be allocated to long fields of different length, so that block fragmentation can be kept to a minimum. This is similar to the *fast file system* in UNIX• 4.2BSD.

The idea of *aggregation* relationships (*complex objects*), has been extensively investigated by researchers at the San Jose IBM Research Laboratory [LORI83], by implementing them as extensions to their System R DBMS. Their implementation involved a *map* for each *complex object* defined, which contains the number of elements and where they can be found. Tuples in relations are assigned system-generated, unique identifiers at the time they come into existence, and two new columns are introduced to model complex objects. These are the

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

"*identifier*" and the "*component of*", which can assume as values the unique identifiers of tuples in the same and/or other relations. The other two types of relationships are modeled using a *reference* column. Note that this new data type is actually a physical *link* across relations at the implementation level and is actually a departure from a true relational model, but its use actually resulted in a performance advantage. Their work has recently materialized in a new implementation based on the relational product SQL/DS [LORI85], with other extensions to handle long transactions and object versioning.

1.4.3. Version Control of Design Data

In [HAYN83], a new data type "*access attribute*" was introduced. Attributes having this particular type may take on values which can be used to access a particular "*instance*" of a relation, by defining an additional "*control*" relation that contains history information. Each instance of any relation is uniquely defined by the relation name and the access attribute in the control relation. Yet another method proposed was to add two columns in each relation, namely "*revin*" and "*revout*". The *revin* column of a tuple represents the revision number at which this tuple was added to the relation and the *revout* contains the revision number at which this tuple became obsolete.

On a more abstract level, a version of a high level object, (a complex object or even a relation), is just another instance of that object. In the Mosaic Object Manager [ATWO85], the most recent version of an object is stored in a fully articulated form, while older versions are stored using a backward differential representation, resulting in a very low storage overhead for older versions. Version paths are allowed to fork off to form *alternatives*, which can be individually addressed using the same version-id scheme as in specifying different versions. Individual versions of high-level objects are used to group mutually consistent sets of related objects, and this provides much finer granularity in objects locking in concurrency control.

Likewise, IBM's enhanced SQL/DS [LORI85] also allows design objects to have different versions, each being represented by another system-generated complex object that holds all necessary bookkeeping information. Alternatives and revisions are modeled by logical *version clusters* that package versions of one design object into groups.

1.4.4. Concurrency Control

As pointed out in [LORI85], one simple approach in solving the problem of long transactions common in engineering data base environments, is to let an engineer *check out* an object from a shared (public) data base and store it in a private data base, leaving a nonvolatile lock in the shared data base. When the design of that object has reached a consistent state, the engineer *checks in* the altered object back into the shared data base, thereby releasing the lock. This is exactly the approach adopted by the extended System R [LORI83]. Recovery is effected, in case of a failure during *check in*, by a log file recording all changes to the public data base when the *checked out* object is returned.

A mechanism for supporting *nested transactions* [KIM84] has been proposed to allow team members involved in the design of the same object to exchange incomplete objects (partial consistency) but it was not implemented because of its sheer complexity and the constraints its protocols impose on the user. Instead, [LORI85] turned their attention to versions and an access control-based cooperation mechanism. Basically, users are assigned to one or more *user groups*. A group may belong to other groups. There is a predefined group, called PUBLIC, that contains all groups as well as users that are not in any other group. Each complex object (versions) is owned by exactly one user or one group, resulting in a number of disjoint *logical data bases*. Users may only access the logical data bases that are owned by themselves or by a group to which they belong directly or indirectly. In this way, versions can be frozen and access to them authorized on a group-oriented (or project-oriented) basis.

1.4.5. Efficiency of Relational DBMS

It was pointed out in [HAYN83] that proper design of a data base, to achieve normalization of the data schema, can minimize data redundancy and result in improved performance. Direct pointer links between tuples in the same relation or between tuples of different relations can be used to speed up data retrieval.

Through their experience in extending INGRES to handle ADTs [STON83], their conclusion was that provision for access by spatial location will substantially reduce search time for geometric data. This can be implemented by defining a spatial "*bin*", or a grid, with an index which is a file containing, for each bin, pointers to all the objects that might overlap that bin. *Extended secondary indices* for columns containing ADTs are also implemented by allowing users to define their own hashing functions, or some functions that will effect the indexing of the ADT, in INGRES.

Yet another approach was presented in [LENZ84] to add data base management functions in the operating system kernel to speed up data access. Due to the fact that most existing DBMS are disk-resident, extensive overhead, in terms of disk I/O, is required when a data base is searched. They defined a "*persistent object*" to be a named collection of bytes under the control of the data management kernel. These "*persistent objects*" are mapped into virtual memory when access to them is requested through a system call. The kernel provides services to create, manipulate and delete a persistent object. It can be considered as a generalization of the *superblock* in the UNIX• file system. Being first implemented in the UNIX• kernel, it has the additional advantage of generalizing advanced facilities such as concurrency control, support of backout and commit, support of a variety of share options, etc. in the kernel for all types of data management activities.

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

1.4.6. Other Products

Some successful (or at least partially successful) products resulting from years of research have already been mentioned: *Mosaic Object Manager* [ATWO85], IBM *System R* [LORI83], IBM *SQL/DS* [LORI85] and the experience of [STON83] in *INGRES*. The following are examples of other attempts not yet mentioned, each having their unique features.

CORD Data System [VU85] is an integrated environment for project control and data management for engineering design data introduced by *CADTEC Corporation*. *CORD* is based upon a relational model and includes desired features such as versioning, concurrency control and access control. *CORD* includes an *Alert* feature which is an "automatic process" invoked by the system based on specific design activities that affect the state of the project. This facility can trigger actions as simple as notification of specific users via mail upon detection of modifications in a project segment, or as complex as initiating user defined processes. In *CORD*, a project can be *promoted* or *demoted* to different phases of the development process. *Alert* can automatically trigger processes to perform the necessary checks, such as running verification tools to check design rules or constraints that are required for the design to pass to the next promotion level.

In [BARA85], INGRES is once again chosen as the fundamental building block on which a modular system for handling design data was created. Instead of simply adding more routines to INGRES itself as in [STON83], their implementation provides the user with two logically separate modules which serve as an additional interface between the user and INGRES (The user can still access INGRES facilities directly, if necessary). The *Data Base Interface* (DBI) facilitates users' access to the public data base, which is maintained by INGRES. It also supports complex objects, concurrency control and constraints enforcement. The second module is the *Local Information Processing Subsystem* (LIPS) that allows a user to map out a subset of the public data base (possibly into local memory) after freezing the consistent set in the DBI. This, as reported, tremendously improved the performance of

INGRES (which is basically disk resident). The user has the choice of putting individual relations defined in his view on disk or in main memory. LIPS and DBI can be considered as subsets of the extended System R previously mentioned. Although LIPS has various features that makes it resemble a DBMS, the biggest difference is that the data LIPS maintains is usually volatile.

Finally, The *IPAD* project (Integrated Programs for Aerospace-Vehicle Design), which began in 1976 at the Boeing Computer Services Company, recently materialized into the EMIS project (Engineering Manufacturing Information Systems) [JOHN85]. *IPIP* (IPAD Information Processor), one of the facilities provided in *EMIS*, currently supports multiple data models, multiple levels of schemas, geometry data, data sets, and concurrent, multi-user, multi-thread access through multiple application interfaces in a distributed environment. *IPIP* supports both the relational and network data models via a single family of data description languages and a single data manipulation language. A data base may be viewed simultaneously through relational, network and hybrid schemas, and a logical schema of one model can be mapped to a schema of another data model freely. Another interesting feature of the IPAD DBMS is its ability to allow users to declare rules and/or constraints on data relationships within and between relations. Apart from allowing the users to turn rule checking off and on, it also supports a tolerance capability through which users may specify an approximation of equality to be used in qualification operations. This is highly desirable since nothing is absolute in an engineering environment.

1.5. Research DBMS and Planned Extensions

1.5.1. A Basic Relational Data Base Management System

This is a simple relational DBMS that the author has built as part of the course work in the Fall, 1984 offering of "*Data Base Implementation*". Through a simple interactive interface, the following operations on the data base are supported:

- **define** <relation> {<attribute> <type>}
- **append** <relation> {<attribute>}
- **display** <relation>
- **project** <relation> [over {<attribute>}] [unique]
- **delete** from <relation> {where <attribute> = <value>}
- **join** <relation1> <relation2> on <attribute1> = <attribute2>
- **update** <attribute> = <value> in <relation> {where <attribute> = <value>}
- **retrieve** <attribute> from <relation> {where <attribute> = <value>}

Each data base under this DBMS is grouped under a subdirectory in the UNIX• file system, together with its associated system relations. A "*r_tbl_ind*" table stores the names of all the relations defined in the data base. Each tuple contains an index to a "*r_tbl_dat*" relation which contains further information of that relation such as the record size of each tuple, the file name under UNIX• that stores this relation, and finally an index to the "*a_tbl*". The "*a_tbl*" records the attributes associated with each relation in the form of a linked-list. This departure from the true relational data model is purely for efficiency and to facilitate garbage collection. Each relation is implemented as two separate files under the UNIX• file system, namely an index file and a data file.

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

The following are general features of this simple DBMS:

(1) All index files, including that for the relation table, are maintained as sorted and sequential files. Retrieval using primary key is performed by reading in the entire index file into a buffer internal to the program and performing a binary search for the key. A hash table might be more efficient in finding one record but it requires reading in the table into virtual memory and sorting it for each display, not to mention the undesirable effect of having to predefine the size of the files.

(2) A binary search either returns the location of where the key was found or where the new record should go if the key was not found. Insertion of a new record in an index file is in effect an insertion sort whereby all records having a key less than that of the new one are first written out to the file, followed by the new record, and finally, the rest of the original file.

(3) All data files are arranged as unsorted sequential records pointed to by pointers in the corresponding index file. Memory freed by delete operations is reused in future append or define operations by keeping a "next free" pointer and an "empty record count" in the header of each data file. Records in the attribute table that pertain to the same relation are linked up by pointers.

(4) The whole data base, including the relation and attribute tables, resides in secondary storage at all times so that the DBMS program can be invoked and terminated repeatedly without affecting the data stored. Any operation that involves the updating or inserting of new records is not performed until the correctness of the data has been checked.

(5) Reasonable error checking has been built into the program. Missing keywords, type mismatch or error in accessing any part of the data base, as well as possible corruption of data files, are reported to the user and the requested operation

aborted. An on-line quick reference of the legitimate commands and their syntax is also incorporated into the program.

The major limitations in this DBMS are as follows:

(1) The only data types supported are that of integer and character strings. To better support user-defined ADTs, more base types should be supported (for example, real numbers).

(2) The "*where*" clause in conditional expressions only support conjunctions. This has to be extended to handle more complex expressions.

1.5.2. Design of Proposed Extensions

The following sections present, in some detail, the proposed extensions to the basic relational data base presented in section 1.5.1.

1.5.2.1. Definition of ADT's and Associated Operations

Support of ADT's involves defining a set of system functions, in addition to those which implement the basic relational algebra, to allow users to define new data types and the new operations associated with each new type. The following new operations are required:

- **define_type**" <type> {<member> <type>}
- **delete_type**" <type>
- **display_type**" [<type>]
- **define_optr**" <optr> <cardinality> <precedence> <left> <right> <result> <file> <func>
- **display_optr**" [<optr>]
- **delete_optr**" <optr>

In addition to the system table *Relations* which stores information on the user-defined relations in the data base, an additional set of system generated and maintained tables are required to store the information describing the user-defined ADT's and their associated

operations. Basically, a *ADTNAME* relation contains the name, number of members and the length of the internal representation of each user-defined ADT. Another *ADTMEMBER* relation is used to keep track of each of the individual members of each ADT and their data types. Abstract data types can be defined as elementary types or as recursively predefined ADT aggregates. Yet another *OPERATORS* relation is updated when a new operation is defined on already defined ADTs to reflect the name, cardinality, precedence and types of the operand(s) and resultant. This particular function of the DBMS also involves compiling the user-defined functions that implement these operations and linking them into the DBMS. Of course, the ideal case is to link them in dynamically at run time.

Furthermore, the command parser has to be modified to recognize these user-defined ADTs and know when and where to invoke the user-defined functions that perform the user-defined operations on these ADTs. Assignment of values (of some other attributes or in external representation) to attributes of equivalent data types should be handled automatically. The internal representation and manipulation of ADT must always be transparent to the user.

1.5.2.2. Arbitrarily Long Text

One instance of ADT that is of importance in engineering design applications is that of arbitrarily long bit and character strings for storing images, graphical and textual descriptions of designs. Instead of requiring the data base designer to define his own ADT to handle long text, a built-in *text* data type will be implemented in the DBMS. The UNIX• file system can be used to serve this purpose but this method uses additional *inodes* of the UNIX• file system unnecessarily. Moreover, the UNIX• file system imposes a upper limit on the maximum number of files permissible on the system. Therefore, a more practical approach is needed. This suggests that a mini file system that resembles the UNIX• file system should be main-

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

tained by the DBMS itself to manage textual information. Instead of imposing a limit on the size of the data space as in the UNIX• file system, data blocks should be linked up as a linked list to allow for unlimited growth.

A file separate from all other relations will be used for storing fixed length records of long text or long strings of bit pattern. A "*text*" data type is internally represented by a long integer which is an index to the starting record in this file. The subsequent records, if any, will be linked up in the form of a linked-list. Records freed from "*deletion*" operation are rechained in the "*free list*" in that file. This prevents large gaps of unusable memory space inside the file. This implementation was adopted to implement the "*a_tbl*" as presented in section 1.5.1 and was found to perform satisfactorily.

1.5.2.3. Design Data Version Control

In order to support versioning and configuration management, an additional system relation is required to store the history for different alternatives as well as the different versions within each. This "*history_tbl*" contains information such as alternative name, version number, the date when it was started, and a description of the characteristics of that version/alternative. The system also has to maintain two system-wide registers to keep track of the current alternative and version that the user is working on at any point in time.

Each tuple in all relations will have (n * v) bytes associated with it to store the information concerning the alternative and version when that tuple is "visible" to the user. n is the maximum number of alternatives of the same design allowable in that particular data base and has to be defined by the user when the data base is created. v is the number of bytes used to store the versioning information and is split into two equal parts. The first is the *revin* which is the version when that tuple was first introduced, and the second part is the *revout* which is the version when that tuple becomes obsolete and should therefore not be

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

((revout <= current version) || (revin > current version))

For example, using one byte for versioning information per alternative makes it possible to have a maximum of 15 different versions for each alternative (versions 0 - 14). Additional storage overhead comes in the form of the additional bytes associated with each tuple, but this is still a significant savings compared to the memory required to duplicate tuples shared by different versions/alternatives that are otherwise identical.

Referring to the description of the basic DBMS in section 1.5.1, each relation in a data base is implemented as a sorted index file containing the primary key together with a randomly ordered data file. The *version bytes* can be tagged onto the entries of the index file and together they form a new primary key that uniquely identifies each tuple in the relation.

Data access procedures will have to be modified to recognize the existence of these *version bytes*. The particular set of *version bytes* to look at in a search operation depends on which alternative is active at that particular moment. Similarly, update procedures will have to modify these special bytes in each tuple to correctly reflect at which version and alternative the update was carried out. Deletion of a particular tuple will be reflected as that tuple being obsolete from the current version onwards (by changing *revout*). The tuple still exists physically in the data base so that it is visible if the user chooses to look at previous versions of the same design. Updates of an attribute within a tuple which was carried over from a previous version will require the duplication of the existing tuple to a new tuple, with *revin* set to the current version number. Thereafter, subsequent updates can be directly performed on this new tuple until another new version is created.

1.5.3. Summary of Planned Extensions

An extended relational data base management system will be implemented under the UNIX• operating systems, with the following capabilities:

- basic data types supported are integers, character strings of predefined length, real numbers, and text of arbitrary length
- support for basic relational algebraic operations such as project, select and join
- provision for user to define ADTs as aggregates of elementary data types and/or recursively defined ADTs
- provision for user to define new operations on already defined ADT by supplying C functions that implement these operators
- ability to maintain different alternatives and/or versions of design data, allowing users to view only the selected version of the current alternative being worked on

1.5.4. Outline of Thesis

Chapter Two contains the functional specifications of the DBMS. Chapter Three presents the architectural backbone of its design, which should give the reader an overview of the whole DBMS. Chapter Four breaks the DBMS down to its separate components and show their design and implementation in detail. Chapter Five describes the test case used to evaluate the DBMS. Finally, Chapter Six draws conclusions from the tests described in Chapter Five and also presents possible future improvements to the DBMS.

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

CHAPTER 2

FUNCTIONAL SPECIFICATIONS

2.1. Operating System and Implementation Language

The UNIX• operating system is becoming more and more widely used in the engineering field both at the application and development levels. Therefore, it is chosen as the target operating system for the development of this extended DBMS. The programming language, C, is the language chosen to implement the DBMS because of its low level byte manipulation capability. This DBMS is written as a utility in the UNIX• universe and is command driven. The following section defines the functional capability of this DBMS.

2.2. Functional Definition of Supported DBMS Commands

2.2.1. Definition of Objects in The Data Base

An object in the data base can be a relation, an Abstract Data Type, or an operator defined by the user. Any such object comes into existence after it is registered to the DBMS by the user using one of the following commands. Like other forms of data in the data base, visibility of such objects are controlled by the version control system in the DBMS.

The command

definerel <rel_name> <att_name att_type>...

defines a new relation with name *<rel_name>*, having attributes with data types defined as pairs of *<att_name att_type>* on the command line. The data type of each attribute must

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

either be a basic pre-defined data type or a previously defined Abstract Data Type. The relation file for the new relation is created and initialized, and a new tuple is added to the "RELATIONS" system relation to reflect the existence of the new relation. The attribute list is stored in the record file "ATTMEMS".

The command

defineadt <adt_name> <adtname adt_type>...

defines a new Abstract Data Type with name *<adt_name>*, which is made up of elements specified as *<adtname adt_type>* pairs. The data type of each element must either be a basic pre-defined data type or a previously defined Abstract Data Type. Upon the validation of the specified data types for its elements, the definition of the new ADT is stored in the relation file "ADTS" and the record file "ADTMEMS" to reflect the existence of the new ADT in the current version of the design data base.

The command

defineop <op_name> <prec> <card> <ltype> <rtype> <restype> <filename>

defines a new operator that acts upon either predefined basic data types or previously defined Abstract Data Types (or a combination of both). *<prec>* defines the precedence of the new operator with respect to other operators already defined in the data base. *<card>* defines the cardinality of the new operator and is either 1 (unary) or 2 (binary). The data types of the left and right operands are defined in *<ltype>* and *<rtype>*. The data type of the result of the operation is specified in *<restype>*. The argument *<filename>* is a pathname of a file in which the user provides assignment statements that make up the exact sequence in performing the operation. Basically, a complex operator on ADT's is broken down into simplier steps that invoke known operators (either pre-defined or user-defined) to act on the operands. Each assignment statement is in the form of "*<identifier>* = *<expression>*". The

keywords "*LEFT*", "*RIGHT*", and "*RESULT*" are used to reference the left operand, right operand, and the result respectively. In addition, temporary identifiers, %1, %2, ..., %9, can be used to store intermediate results of evaluation. The data types of these temporary identifiers do not have to be defined; they take on the data type of the result of an expression when they are first set. The left hand side of the last assignment statement in the file must be "*RESULT*".

2.2.2. Relational Algebraic Commands

The following commands allow the user to input design data into the data base, as well as to specify queries on the data base using standard relational algebraic operators such as *project*, *view* and *join*. A more detailed functional specification of each available command follows.

The command

append <rel_name> <att_value>...

is used to add a new tuple to the relation *<rel_name>*. The number of attribute values *<att_value>* must equal the number of attributes defined for that relation. For each attribute, the *<att_value>* must conform to the expected external representation of the corresponding data type.

The command

display [relations|adts|operators|versions|<rel_name>]

either displays the content of an entire relation by the name *<rel_name>*, or displays information in the system relations. "*Display relations*" shows all the relations defined in the DBMS in the current context, together with a list of their attributes and data types. "*Display adts*" reveals all the Abstract Data Types that the user has defined, together with a list of their ele-

ments and their data types. "*Display operators*" gives the definition of all the user-defined operators in their textual form, together with other information about those operators such as their precedence, cardinality, operands and result data types. "*Display versions*" provides a summary of all the defined versions currently recorded in the design data base, each being accompanied by the time when that version was started and a textual description of each.

The command

project <rel_name> [over <att_name>...]

is similar to the command *display* but instead of showing all the attributes, *project* only shows the values of the attributes specified in the attribute list in the "*over*" clause. The order of the columns in the output is specified in the "over" clause. Notice that the "*over*" clause is optional. If the "*over*" clause is omitted, this command reverts to a *display*.

The command

delete [from] <rel_name> [where <condition>]

is used to delete tuples from a relation specified in *<rel_name>* which satisfied the condition in the "*where*" clause. The *<condition>* in the "*where*" clause is specified as an expression which, when evaluated, should yield a result of the type INT (integer). If the optional "*where*" clause is left out, the entire relation will be removed in the current version of the design. If a tuple or a relation is carried over from a previous version or another alternative, its deletion in the current version does not affect its existence in previous versions or other alternatives.

The command

retrieve <att_list> from <rel_name> [where <condition>]

is a general retrieval command that retrieves the values of the attributes (or elements of the

attributes, if they are defined as ADT's) specified in the *<att_list>* from the relation *<rel_name>*. The optional "*where*" clause limits the output to those tuples that satisfy the *<condition>*.

The command

view <assignlist> from <rel_name> [where <expr_list>] [into <new_rel_name>]

is a variant of the *retrieve* command which allows users to obtain an alternative *view* of the data in the data base. Instead of displaying the retrieved data in the form that they are stored, the user can specify a list of assignment statements (*<assignlist>*) to transform the data through some additional operations to give new meaning to the retrieved data. The optional "*where*" clause serves the usual function of limiting the output to certain tuples in the relation *<rel_name>* that satisfy the *<condition>*. The optional "*into*" clause instructs the DBMS to generate a new relation by the name of *<new_rel_name>* to contain the result of the *view* operation. The facility allows the user to dynamically create a new "*view*" of the data in the design data base and treat it as any other existing relations.

The command

join <rel1_name> <rel2_name> on <att in rel1> = <att in rel2>

performs an *equi-join* on the relations *<rel1_name>* and *<rel2_name>* by checking for equivalence of the *<att in rel1>* and *<att in rel2>* columns in the two relations.

The command

update <rel> <assignlist> [where <condition>]

updates the values of attributes specified in the assignment list *<assignlist>* in the relation *<rel>*. The new values are the result of evaluation of the right hand side of the assignment statements in the *<assignlist>*. If the optional "*where*" clause is omitted, every visible tuple in

the relation will be updated, otherwise only those tuples that satisfy the *<condition>* will be updated. If the affected tuple is carried over from previous versions or another alternative, a duplicate tuple is made and the old tuple is marked obsolete while the actual update is performed on the duplicated tuple. The duplicated tuple is only visible starting from the current version but not in older versions.

2.2.3. Versioning Control Commands

Alternatives are defined to be different instances of the same design which are meant to be worked on simultaneously. The maximum number of design alternatives is fixed at compilation time of the DBMS. Currently, the maximum number of alternatives is set at four.

Different versions of the same design are allowable in each alternative. A particular version of a design is built upon older versions (with the exception of version zero). When a new version is defined, the state of the design data base in the previous version is frozen and alterations to the new version will only be reflected in the new version and will not affect the state of affairs in the older versions. The maximum number of versions in each alternative is again pre-defined at compile time. Currently, this value is set at sixteen.

A particular version in any alternative is identified by the combination of its alternative and version numbers in the form *<alternative_no>.<version_no>*. Maintenance of versioning information is handled by the **DB Subsystem** (refer to Chapter Three for a description of the **DB Subsystem**) and is transparent to the user as well as the higher level modules. At any one point in time, one and only one combination of alternative and version is active. The user can only see the data that is supposed to be visible in the current version. The only way of viewing data in another version is to change the current context to that version.

The command

chalt <alt_no>

changes the currently active alternative to *<atl_no>* to allow the designer to freely switch between different alternatives of the design.

The command

newalt

creates a new alternative in the design data base and carries over all relevant data in the design data base that are currently visible. The new version number will start at zero for the new alternative and the new alternative will become the currently active alternative.

The command

chvers <version_no>

changes the currently active version to *<version_no>* to reflect a different state of the design data base.

The command

newvers

freezes the current version and changes the context to the next higher version.

The command

showversion

displays the currently active alternative and version numbers in the form *<alternative_no>.<version_no>*.

2.2.4. Miscellaneous Commands

The following commands serve to provide the user with a more user-friendly environment.

The command

setvar in <rel> $<var_name> = <expr> where <condition>

allows the user to set a variable named *<var_name>* (arbitrary) to the result of the evaluation of the *<expr>*. Basically, this command is used to store the result of a *retrieve* operation in a temporary repository for future use (instead of simply displaying it on the screen and then throwing it away). The data type of the variable will be automatically set to that of the result of the expression *<expr>*. Once set, the content of a variable can be referenced and used in later operations as an operand in expressions in "*where*" clauses or assignment statements.

The command

showvar [$<var_name>]

shows the value of the variable *$<var_name>* in its external representation. If no argument is provided, the values of all the variables that are previously set will be displayed.

The command

unsetvar $<var_name>

makes the variable *$<var_name>* extinct and frees the internal variable table entry originally occupied by that variable. That entry can then be reused for another variable. The internal variable table size is currently set at 50.

The command

!!<commands>

appends *<commands>* to the last executed command to form a new command to be executed by the system. If *<commands>* is null, the last executed command is executed again. This is basically provided to save typing on the users' part.

The command

!<commands>

invokes a shell escape to the UNIX• shell where the *<commands>* are performed. This allows the user to make the best use of the facilities provided by the UNIX• operating system.

The command

?

prints out a one page concise help manual that summaries all the available commands in this DBMS.

The command

< filename

instructs the system to read in command lines from the file by the name of *filename*. Each line read in by the DBMS is echoed to the output and the user is prompted for a confirmation before the command is actually performed. This facility allows the users to create command files for frequently repeated operations.

The command

<< filename

is identical to the above command except that no confirmation is required from the user. Command lines are read in and directly executed.

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

The command

> filename

writes out the last executed command to the file named *filename*. If the file already exists, the command is appended to the end of the file; otherwise the file is created and the command is appended.

CHAPTER 3

ARCHITECTURAL DESIGN

3.1. Introduction to Design of Research DBMS

This chapter starts with an overview of the structural design of the Data Base Management System (section 3.2). The DBMS is broken up into three distinct levels. At the highest level, there is the user interface to interpret user input and queries. In the middle lies the center of control, the "DB Manager", who takes up the task of delegating different tasks to different sections in the organization. At the lowest level, The DB "Subsystem" will interface with the UNIX• file system to store and retrieve data in the data base which is maintained in secondary memory at all times.

Section 3.3 describes the interfaces between the DBMS and its environment. Section 3.4 shows the internal data and control flow of the whole system.

3.2. Overview of Research DBMS

Figure 3.1 shows an overview of the structure of the Data Base Management System. The "User Interface" is responsible for parsing user's commands to query the data base. Early error detection is attempted to check for obvious syntactical errors in the command line. The "Command Line Parser" parses the command line up into tokens and passes it down to the DB Manager for further processing.

The "DB Manager" examines the parsed command line to determine which of its agents should be called to handle the request. For example, the "Operator Manager" maintains

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

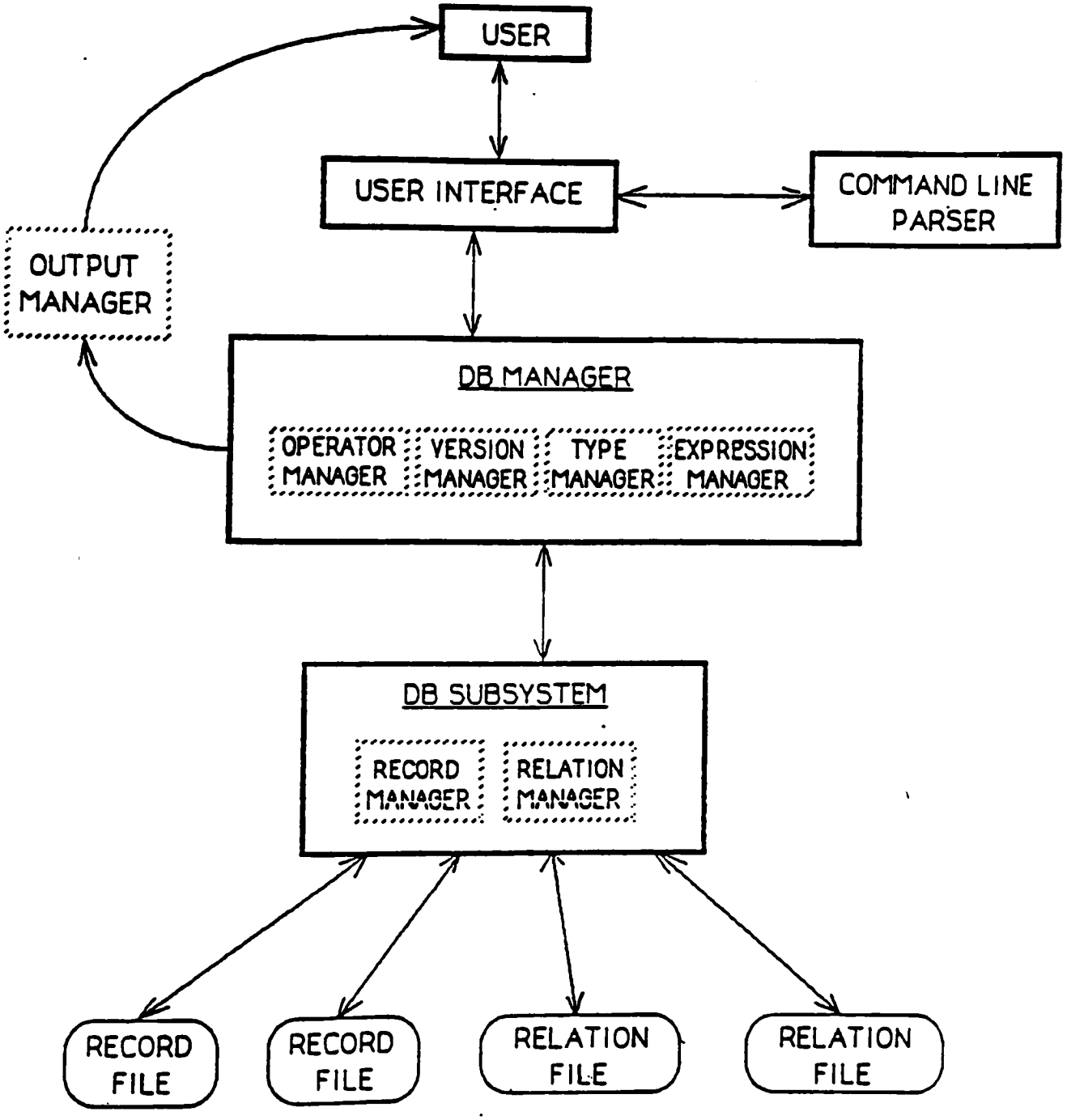

Figure 3.1. Overview of DBMS

information pertinent to basic operators such as addition and multiplication, as well as user-defined operations defined on Abstract Data Types. This particular agent is responsible for the identification and evaluation of basic pre-defined operators as well as user-defined operators on ADT's. The "Version Manager" keeps track of the current alternative and version of the design data base and determines which parts of the design data base should be visible in the current context. The "Type Manager" is responsible for identifying both basic, predefined data types as well as user-defined Abstract Data Types. The "Expression Manager" provides support for parsing and evaluation of mathematical expressions.

Information pertaining to the request is gathered as needed by the "DB Manager" from its agents. Control is then passed down to the "DB Subsystem" to extract the "raw" data from the relations. The "DB Manager" then invokes the appropriate utilities under its control to examine and format the "raw" data. If the data matches the conditions set forth by the request, it is sent to the "Output Manager" and queued for output.

The "DB Subsystem" handles all the lowest level data storage and retrieval by directly interfacing with the UNIX• file system. Storage format and retrieval methods are made transparent to the higher level routines so they can view the data at a more abstract level. Design data version control is also handled by the "DB Subsystem" and its work in this respect is virtually invisible to the higher level managers.

Detail description of various sectors of the DBMS will be presented in Chapter Four.

3.3. User Interfaces

This DBMS is command driven. The user can interface with the DBMS by entering on-line commands, or may choose to group commands in a file and execute them as a batch. A "quick reference" manual is included to assist the user in his/her communication with the DBMS. Furthermore, access to the the UNIX• operating system is provided. Helpful

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

UNIX• features include shell escapes and the ability of the DBMS to read commands from a file or storing user query commands in a file for future use. Chapter Two contains detailed functional specifications of all the available commands supported by the DBMS.

In order to support the definition of ADT's upon previously defined ADT's, the notion of *list* and *sublist* is introduced to the command parser. Normally, upon input of data from the user terminal, the space character is considered to be the field separator. However, the user can override this by enclosing any string of characters within matching curly brackets ({ , }). The command line parser simply parses each command line into tokens or lists of tokens and passes them back to the "DB Manager". Curly brackets in the input can be nested to form *sublists* within a list. Each time a string is parsed, the outmost pair of matching brackets will be stripped to reveal the list inside them, so that sublists are broken up only when they need to be, but not before that.

This notation for the representation of *lists* and *sublists* is maintained across the input and output boundaries of the DBMS. Data items that represent ADT's are enclosed within curly brackets when they are presented to the user.

All relations are stored as individual files in the UNIX• file system. The DBMS interfaces with the UNIX• file system through the "DB Subsystem", which is the only module in the whole DBMS that has direct access to the "raw" data stored in the data files. The "DB Subsystem" interfaces with the UNIX• file system through low level system calls such as "seek", "read" and "write". This restriction of direct access to the "raw" data to only one module makes this interface much cleaner and hides the details of version control from the higher level modules.

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

3.4. Data and Control Flow

The "DB Manager" communicates with its agents and the "DB Subsystem" mainly through global data structures which have fixed formats. These buffers are passed through the whole system and are modified as appropriate.

For example, a particular relation comes into view when its relation buffer (_RELATIONBUF) is created by the "DB Manager". Information about that relation is then gathered by different *managers* as this buffer passes through their *departments*. For example, a _RELATIONBUF structure is defined as follows:

```
typedef struct {
        char        rel_name[R_NAME_LEN];
        char        rel_fname[F_NAME_LEN];
        int         rel_fd;
        int         curindex;
        int         next_free;
        int         empty_records;
        int         tuplesize;
        char        *data;
} _RELATIONBUF;
```

The "DB Manager" first allocates dynamic memory for this data structure and fills in the *rel_name* field, which contains the logical name of the relation. The structure is then passed down to the "DB Subsystem" to look for a relation by that name. If a relation exists by that name, the actual file name under the UNIX• file system that is used to store this relation is placed in *rel_fname*. The field *tuplesize* will then take on a value which is the number of bytes used for the internal representation of one tuple in that relation. This "*relationbuf*" is then passed back to the "DB Manager".

Dynamic memory is then allocated for the *data* field by the "DB Manager". This *data* element in the structure is used as a repository for receiving data from the "DB Subsystem" when data from that relation is actually retrieved from the relation file. When the relation file is opened by the function call "*openrelfile*", the rest of the information in the data structure is

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

filled in by the appropriate functions in the *relation package*. The "*curindex*" is used to keep track of the position of the current tuple in that relation. The internal function, "*getnexttuple*", uses this "*relationbuf*" structure to trasport the data retrieved from the next physical tuple in a relation. The "*next_free*" and "*empty_records*" fields are used only by the "DB Subsystem" for garbage collection purposes and are usually not the concern of higher level modules. Upon the completion of a transaction, the memory claimed by these buffers can be freed and reused. Appendix A contains a complete list of all the system buffer structures used globally in the DBMS.

A relation has to be explicitly *opened* by the *openrelfile* function call before tuples can be retrieved from that relation. However, there are certain system-generated and system-maintained relation/record files that are opened when the system is initially started up and are kept opened until the program is terminated. The following is a list of these system relation/record files and their content:

RELATIONS	- User-defined relations
ATTRIBUTES	- Attribute lists for each user-defined relation
ADTS	- User-defined Abstract Data Types
ADTMEMS	- Lists of members and their data types for each ADT defined
TEXTRECORD	- Record file for Textual information
VERSIONS	- Alternatives and versions defined
ADTOPS	- Operators defined on ADT's by users
ADTOPATTS	- Attribute lists for each ADT operator
ADTOPSTATS	- Statement lists for each ADT operator

The main purpose for maintaining these relation/record files opened at all times is to improve the performance of the DBMS. All "raw" data of user-defined relations, as well as system-generated and system-maintained relation/record files are kept in secondary memory and read in only when they are needed.

CHAPTER 4

MODULAR DESIGN OF DBMS

4.1. Introduction to Modular Design of DBMS

This chapter presents the DBMS detail design and describes the implementation of each module. The "DB Manager" has already been introduced as the center of data and control flow in the DBMS (Chapter Three) and will not be considered here. Section 4.2 explains in more details the major parts of the "DB Subsystem" and their functions. Section 4.3 details the implementation in this DBMS for supporting ADT's and their operations. Identification and handling of different data types in the data base is explained in section 4.4. Section 4.5 describes the mechanism for parsing, storing and evaluation of expressions such as those in "*where*" clauses and assignment statements. Finally, the function of the "Output Manager" will be introduced in section 4.6.

4.2. Data Base Subsystem

The "DB Subsystem" is the lowest level module that has sole access to the "raw" data. Its major components are broken down into the following subsections.

4.2.1. Relation Package

There are only two formats of data storage visible to the data base designer. The first is *relations*, whichs consist of unordered, fix-sized tuples. The second is *records*, which is made up of any number of ordered, fixed-sized data blocks. This will be described in more details in subsection 4.2.2.

The *relation package* is responsible for the storage and retrieval of data stored in the

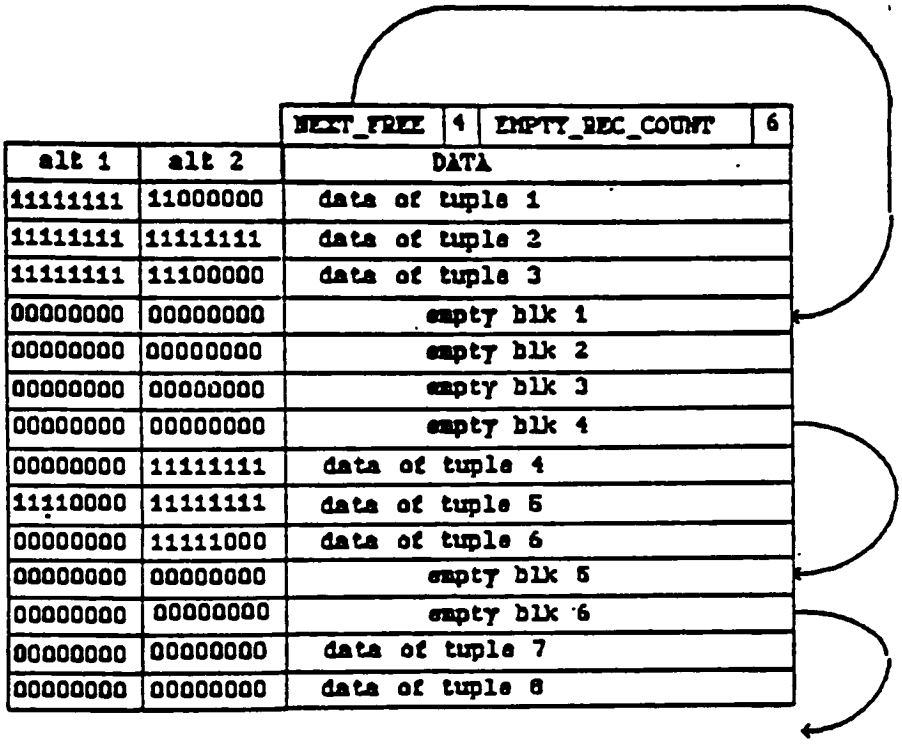

Figure 4.1. Relation File Structure

form of *relations*. A relation is stored as a data file under the UNIX• operating system. The first eight bytes of the file contain information useful for garbage collection when tuples are deleted. The rest of the data is arranged as fixed-sized blocks. Figure 4.1 shows the structure of a relation file. The first eight bytes of a relation file is actually used to store two integers, "NEXT_FREE" and "EMPTY_RECORDS". "NEXT_FREE" is a pointer to the beginning of a linked-list of free blocks for new tuples, and "EMPTY_REC_COUNT" is a count of how many empty blocks are embedded within the file. Empty blocks result when tuples are physically deleted from the relation file (when it no longer exists in any version). The "DB Subsystem" uses the "NEXT_FREE" and "EMPTY_RECORDS" to maintain a

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

linked-list of available free blocks to be used when new tuples are to be added to the relation file.

Design data versioning is handled within this package by tagging each tuple in each relation (including the system relations) with some number of *version masks*. Every access to the relation files calls for the examination of the appropriate *version mask* associated with each relation tuple in the "raw" relation file. Higher level modules only *see* the tuples that are active in the current context governed by the currently active alternative and version of the design. More detailed description of the maintenance of *version masks* can be found in section 4.2.3.

This *relation package* provides facilities for initializing the data files for a newly created relation, as well as other more general facilities for storing and retrieving tuples in that relation. As mentioned before, higher level managers only see tuples in relations that are supposed to be *visible* in the current alternative and version.

The *relation package* communicates with the outside world through a globally defined data structure called _RELATIONBUF. The definition of a _RELATIONBUF is presented below:

```
typedef struct {
        char        rel_name[R_NAME_LEN];
        char        rel_fname[F_NAME_LEN];
        int         rel_fd;
        int         curindex;
        int         next_free;
        int         empty_records;
        int         tuplesize;
        char        *data;
} _RELATIONBUF;
```

The field "rel_name" contains the logical name of the relation as known to the designer. The character string "rel_fname" contains the UNIX• file name of the plain file that is used to store the information in the relation. The file descriptor "rel_fd" is set when the relation file

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

is opened by the "openrelfile" function call. The pointer "curindex" keeps track of the current *cursor* position in the relation file. Only sequential search is supported on relations. The "next_free" and "empty_records" counters are maintained by the *relation package* for garbage collection purposes. The actual size of one tuple in the relation is stored in "tuplesize". Each time a tuple is requested through a "getnexttuple" function call, the data of the next tuple is mapped into the memory space referenced by the "data" pointer. If there are no more tuples in the relation to satisfy the request, an error is returned to the caller. A detailed list of the functions available in this package is available in Appendix B.

4.2.2. Record Package

Records are made up of an unlimited number of fixed-sized blocks and are used to store information of variable length. The support of the predefined type "TEXT" uses *records* to store character strings or bitmaps of arbitrary length.

Blocks in a *record* belonging to a certain *record file* are maintained as a linked-list of fixed-sized records and are usually referenced by the index of their first block in the *record file*. The "Record Package" provides the facilities to start a new record and append blocks to it in a sequential order. It also allows the opening of a *record* and the sequential retrieval of data blocks in the order they were originally stored. Like the *relation package*, the *record package* communicates with the outside world through a globally defined data structure as shown bellow:

```
typedef struct {
    char    rec_fname[F_NAME_LEN];  /* file name of record file      */
    int     rec_fd;                 /* file descriptor of rec_file   */
    int     rec_no;                 /* index to starting block       */
    int     tblk;                   /* total no of blocks in record  */
    int     nblk;                   /* no of blocks already read in  */
    int     next_free;              /* next block in free list       */
    int     empty_records;          /* count of free blocks */        */
    int     curblk;                 /* current block in record file  */
    int     blocksize;              /* size in bytes of each block   */
    char    *data;                  /* area for data transfer        */
} _RECORDBUF;
```

The index "rec_no" is filled in by higher level functions and is an index to the first data block in a record inside the record file. A record has to be opened with the "openrecord" function call before its data blocks can be retrieved. After a record is opened, each successive function call to "getnextblock" will cause the next data block in that record to be placed in "data", until the data blocks in the record are exhausted.

Figure 4.2 shows the typical structure of a *record file*. Garbage collection of freed blocks is again effected through a linked-list of free blocks. Again, the "Record Package" hides the underlying complexity of "raw" data manipulation from the higher level utilities that uses this facility.

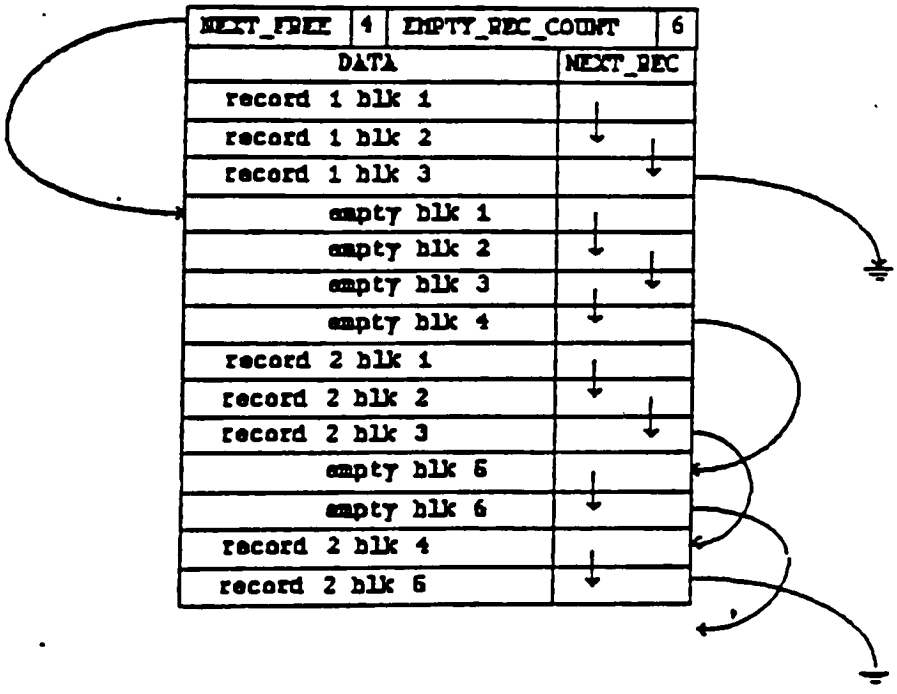

Figure 4.2. Record File Structure

Since record files are only repositories for storing data which is indexed by pointers contained in relation files, the notion of different alternatives and versions is unknown to this package. Functions in the *relation package* is responsible for freeing the records as needed when they delete tuples.

4.2.3. Design Data Version Control

One major function performed by the "DB Subsystem" is version control of the design data in the data base. The actual mechanism for maintaining different versions in the data base is transparent to the higher level modules and the user.

As mentioned before, each tuple in each relation is stored together with a number of *version masks*. Referring to figure 4.1, which shows the typical structure of a relation file, each *version mask* is a bit string which dictates whether a particular tuple should be visible in the current version and alternative. The maximum number of alternatives and versions allowable in each relation in the data base is coded into the DBMS and can only be changed by recompilation of the DBMS program. Bits in the *version mask* for each alternative are set to indicate that the tuple should be visible, and is cleared if otherwise. The most significant bit corresponds to version zero of the corresponding alternative and so on. For example, tuple number 3 as shown in figure 4.1 is only visible in versions zero through two in alternative 2, but it is visible in all versions in alternative 1.

Initially, when a tuple is added, all bits corresponding to the current version in the current alternative and upwards are set to signify when the tuple was added. Deletion of a tuple corresponds to the clearing of all bits in the currently active *version mask* from the current version position upwards (to the maximum version). This means that the tuple will no longer be visible in the current version, but will become visible again if the previous version of that design is reviewed. A tuple still physically exists in the design data base as long as it is visible in some versions and alternatives in the design data base.

Upon deletion, the space occupied by the deleted tuple is freed only if it does not exist in any known version in any alternative defined in the design data base. In this case, that space is returned to the *free list* maintained by the "DB Subsystem" and can be reused the next time a new tuple is added. Creating a new version in a particular alternative has the effect of freezing the instance of the current design so that further changes made to the design will not affect the state of affairs in the previous version. On the other hand, creating a new alternative in the design data base causes the current state of the design to be effectively duplicated in the design data base, and the version number in the new alternative is reset to zero.

4.3. Abstract Data Type Package

Definition of an ADT is stored in the **ADTS** table which consists of the character name for each ADT, the number of bytes required to store it internally, and a pointer to a record in the **ADTMEMS** record file. This record contains the name, type and size of each individual member of the ADT. This departure from the true relational approach is adopted for performance considerations.

An ADT has to be defined before it can be used to define attribute types in a relation. An ADT can be defined as an aggregation of basic pre-defined data types such as integer and floating point numbers, or a combination of basic types and previously defined ADT's. After an ADT has been defined, a data item which has a data type equal to that ADT can be treated and manipulated as a single entity. The individual data items of any ADT are stored internally as a combination of basic pre-defined data types and are formatted to *lists* and *sub-lists* when they are presented to the user. Elements that are part of an ADT can be referenced using the syntax of *<ADT name>.<element name>*.

Users can define new operators on previously defined ADT's. A system-generated and maintained relation, defined by the **ADTOPS**, is used to store the information of all user-defined operators on ADT's. This table contains the following fields:

adtopname	- character name of the operator
precedence	- precedence of the operator
cardinality	- unary or binary
lopnd	- data type of left operand
ropnd	- data type of right operand
resulttype	- data type of the result of operation
text_rec	- pointer to the textual definition of the operator
stat_rec	- pointer to the internal representation of its operation
att_rec	- pointer to a list of attributes used for this operation

The user (data base designer) puts the definition of an operation on ADT's in a file as a sequence of assignment statements. Ten pseudo attributes, namely %0 to %9, can be used to store intermediate results of computation during the process of evaluating the final result of the operation. Upon definition of the new operator, these assignment statements are parsed to check for the compatibility of data types. Left and right operands, and the result of the operation are referenced by keywords "LEFT", "RIGHT", and "RESULT" respectively in the ADT operator definition file. Each assignment statement is parsed and transformed into a post-fix expression and the relative offsets of each element in the expression is calculated. These post-fix expressions are stored for future use when that operator is invoked.

As an example, assume that a type that represents a rectangle is defined as

DEFINEADT RECT UL CORD LL CORD UR CORD LR CORD

where the ADT "CORD" is defined by

DEFINEADT CORD X FLOAT Y FLOAT

This defines a type "CORD" (coordinate) as consisting of a pair of floating point numbers that represents the (x,y) coordinates of a point in a two-dimensional space. A type "RECT" is defined to be an aggregate of four coordinate points, which happens to be the four corners of the rectangle.

Now, the operator "LARGERTHAN" that operates on two "RECT" might be defined as follows

DEFINEOP LARGERTHAN 5 2 RECT RECT INT LARGERTHAN

where the precedence level of the operator "LARGERTHAN" is defined to be 5 and the car-

dinality is defined to be "binary". The left and right operands of the operator are both of type "RECT" and the result should evaluate to an integer (boolean). The detailed definition of how to perform this operation is found in the file named "LARGERTHAN" in the current working directory.

The file "LARGERTHAN" might contain the following definition of the operation:

```
%0 = ( LEFT.LR.X - LEFT.LL.X ) * ( LEFT.UR.Y - LEFT.LR.Y )
%1 = ( RIGHT.LR.X - RIGHT.LL.X ) * ( RIGHT.UR.Y - RIGHT.LR.Y )
RESULT = %0 > %1
```

where the pseudo attribute %0 is set to the area of the rectangle specified in the left operand and %1 is set to the area of the rectangle specified in the right operand. Finally, the area of the left rectangle is compared to the right rectangle to determine if it is larger than the right rectangle.

Definitions of previously defined operators can be reviewed using the "DISPLAY" command with "OPERATORS" as its argument. The textual form of the definition is stored as a *TEXT* record upon its definition for this purpose.

An alternative method of handling user-defined operators is to allow users to write their own C-functions that implement the operations required, and to link them in dynamically at run-time. This, however, is difficult to implement under the UNIX• operating system since it does not provide any facilities to perform dynamic loading of additional modules to an already running process. The major effort in this implementation is the need to write a separate loader to overcome that problem. Nevertheless, this alternative does offer some distinctive advantages over the current implementation in that it gives the user more flexibility. For example, the user can write a function that implements a certain operation on certain ADT's and at the same time performs tasks not directly on the data itself, such as invoking other UNIX• utilities if certain condition in the data base is satisfied.

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

4.4. Data Type Manager

The "Type Manager" maintains its data base to store the definition of basic predefined data types as well as user-defined ADT's. It is responsible for identifying various data types and for looking up the system relation ADTS for the elements of user-defined ADT's. Given an identifier, the "Type Manager" is also responsible for calculating the byte-offset from the beginning of the tuple for the identifier and then returning its data type. This offset value is used later to calculate the actual address of the identifier.

All data types are internally enumerated and the "Type Manager" is equipped to recognize any data type by its character name as well as its internal enumerated ID. Basic data types supported are *integer*, *floating point numbers*, *character strings* (of fixed length) and *text*. Data of type *text* is internally stored as an index in the relation, and this index is used to retrieve the actual data stored in the "TEXT" record file.

For user-defined ADT's, the "Type Manager" stores their definition in the "ADTS" relation and "ADTMEMS" record files. The attributes of the "ADTS" relation are *ADTname*, *ADTsize*, and *ADT attributes pointer*. The *ADT attributes pointer* is used to index into the "ADTMEMS" record file for a list of the elements and their types for that ADT.

4.5. Expression Package

Expressions may occur in "*where*" clauses as well as assignment statements in the *update* command line. Expressions are allowed to utilize the full range of basic operators as well as previously user-defined operators on ADT's.

A set of 14 basic operators are predefined for basic data types. These are:

+	Addition of INT or FLOAT
-	Subtraction of INT or FLOAT
~	Negation on INT or FLOAT
*	Multiplication of INT or FLOAT
/	Division of INT or FLOAT
**	Exponentiation of INT or FLOAT
&	Logical AND on INT
|	Logical OR on INT

=, >, >=, <=, != Comparison operators on INT, FLOAT or CHAR

The "Operator Manager" is responsible for determining if a certain token being parsed is an operator. It first looks up its internal "BASEOP" table and if no match is found, will look up the user-defined operators relation, ADTOPS. It is also responsible for verifying the data types for the operands as required for the particular operation to be performed on them. Information on a recognized operator is placed in a "_OPBUF" structure supplied by the caller. This structure is used to store the information pertaining to the specified operator such as its cardinality, precedence (used for parsing an expression), the data types of its operands and so on.

To speed up the execution of expression evaluation, definitions of user-defined operators are cached into the DBMS's virtual memory once it is identified to be a legitimate operator. A usage count for each operator in the cache is incremented each time it is used. When the operator cache gets full, the one that is least used will be preempted to make room for new operators.

The action of performing an operation on its operands is also handled here. Basic operations on basic data types are straight forward and will not be detailed here. More complex operations on user-defined ADT's are handled differently (see section 4.3).

Expressions are parsed within this package into their *post-fix* form, which is suitable for storage as well as evaluation. Identifiers in an expression have to correspond to some attributes (or its elements if it is an ADT) in the currently active relation. The byte-offset of each identifier is calculated by the "Type Manager" when the expression is parsed. Memory is dynamically allocated to each operator or user-defined variable in the *post-fix* expression to hold the result of that operation. Constants in expressions are transformed to their internal representation and stored with the expression when the expression is parsed.

Figure 4.3 shows the internal data structure of the expression (C + ~ D - CONST INT 3). The expression is scanned from left to right when it is evaluated. If the token is a

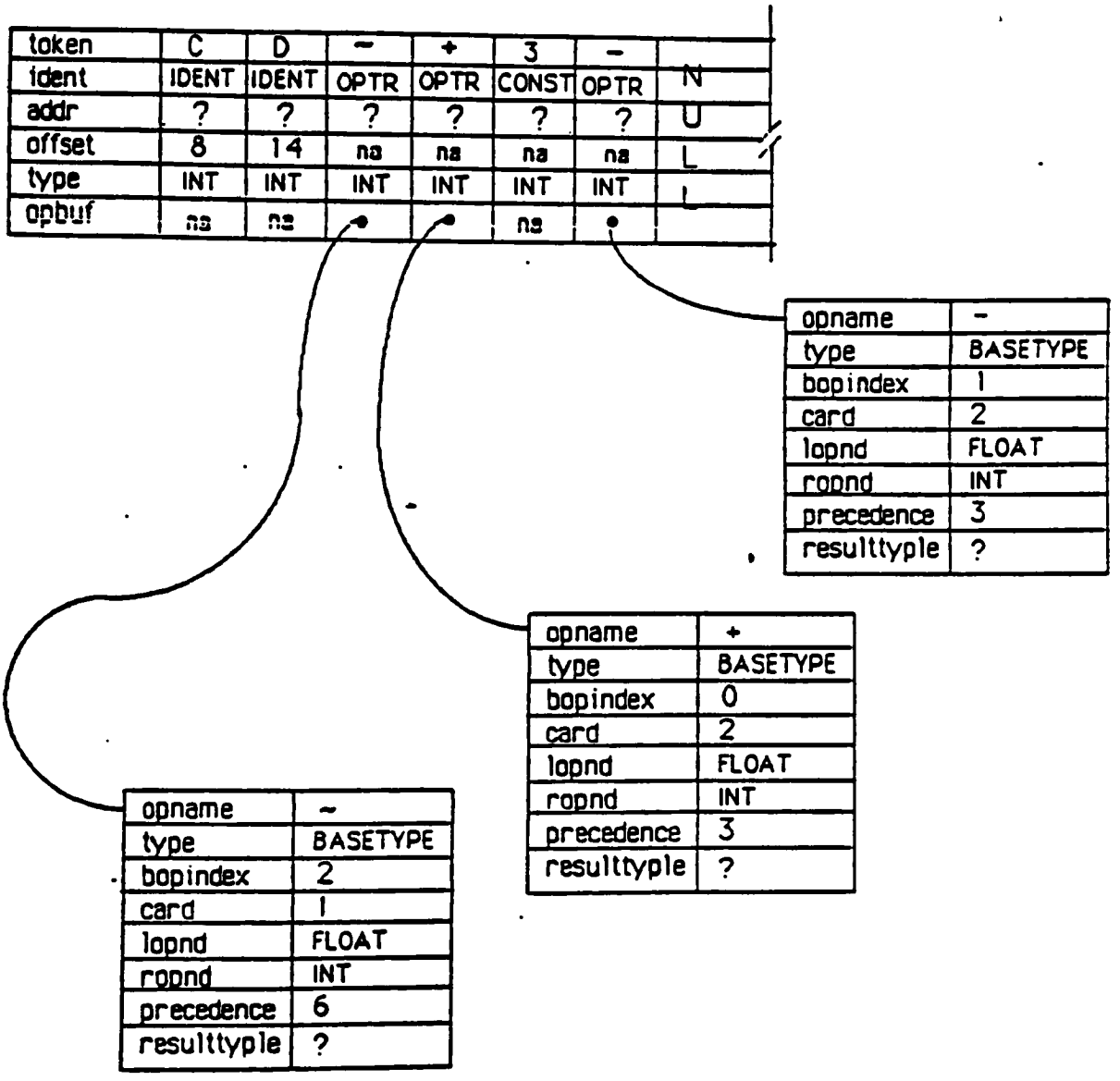

Figure 4.3. Structure of An Expression

constant, its address will be fixed and it is therefore simply pushed on top of the stack. If the token represents an operand but not a constant, its actual address is calculated from the

"offset" field and stored in the "addr" field. The operand is then pushed onto the stack. When an operator is encountered, the appropriate number of operands are popped from the stack and the operation performed on them. Then, the result is placed in the "addr" field of the operator. The result of the evaluation is finally copied into the address space provided by the caller.

4.6. Output Utilities of DBMS

The main function of the "Output Manager" is to convert the result of a data retrieval request into its external representation, and then format it into multi-columns.

An output queue is maintained by the "Output Manager" to store output data until a screenful of data is queued. When one screenful is collected, the output queue is flushed to the user's terminal. This maintains a one screen look-ahead for data retrieval so that the DBMS continues its search in the data base while the user is assimilating the last screenful of output. This reduces the average delay for long searches. Figure 4.4 shows the structure of the output queue. Memory is dynamically allocated for output data to form a linked-list of output records and is freed when the output queue is flushed.

When the output queue is first initialized, the "Output Manager" receives an array of _ATTBUF structures that define the data items which need to be displayed. This array also contains the data type and the relative offset of each data item to be printed. When the "Output Manager" receives a request to queue output to the output queue, it extracts the data items specified in the array and transforms the data into its external representation. A running maximum of the width of each data item in its external representation is maintained internally by the "Output Manager" in order to format the output into multiple columns when the output queue is flushed.

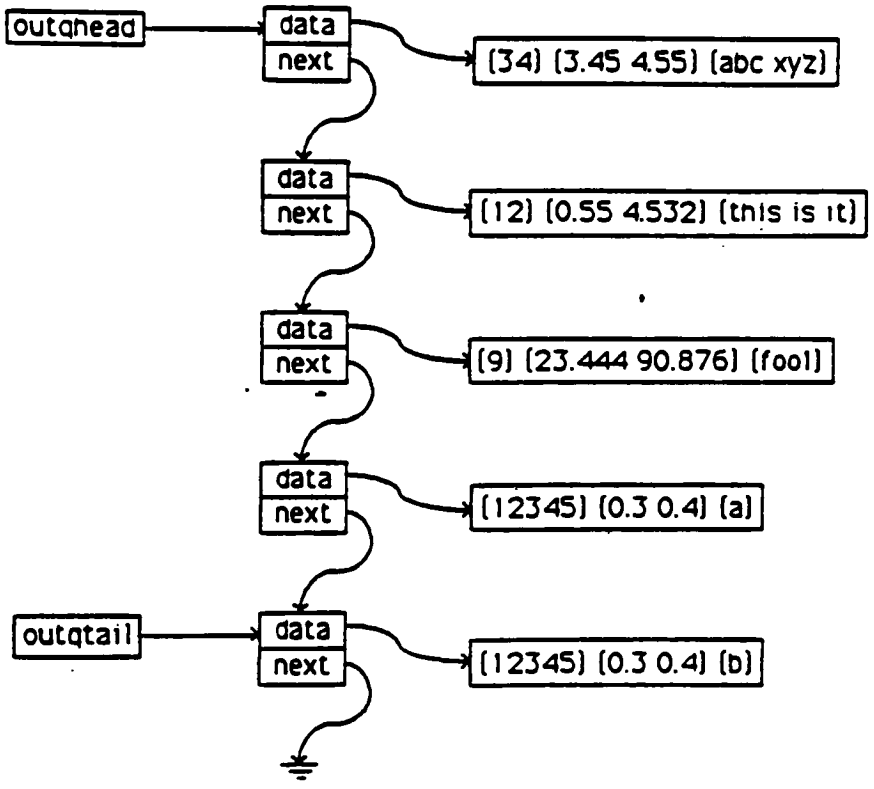

Figure 4.4 Structure of Output Queue

CHAPTER 5

TEST CASE FOR DBMS

5.1. Graphical Representation of DBMS Structure

As a test case, a data schema is designed for the DBMS to store part of the graphical representation of the overview of the DBMS. Figure 3.1 in Chapter Three is simplified and placed in a two-dimensional plane as shown in figure 5.1.

5.2. Data Schema of Research DBMS

In order to model the structure shown in figure 6.1, the following data types are defined.

```
defineadt boxdprops
        linethickness   float
        shading         int
defineadt cord
        x       float
        y       float
defineadt vector
        x_inc   float
        y_inc   float
defineadt line
        start   cord
        end     cord
defineadt box
        title   char    31
        ll      cord
        width   float
        height  float
        displayprops    boxdprops
```

The ADT "*boxdprops*" stores the display properties of a *BOX*. The "*linethickness*" represents the line thickness of the boundary of the *BOX* and the "*shading*" is a code for the kind of shading to be performed inside the *BOX*. Both "*cord*" (coordinate) and "*vector*" consist of two elements which are floating point numbers. The data type "*line*" is an example of

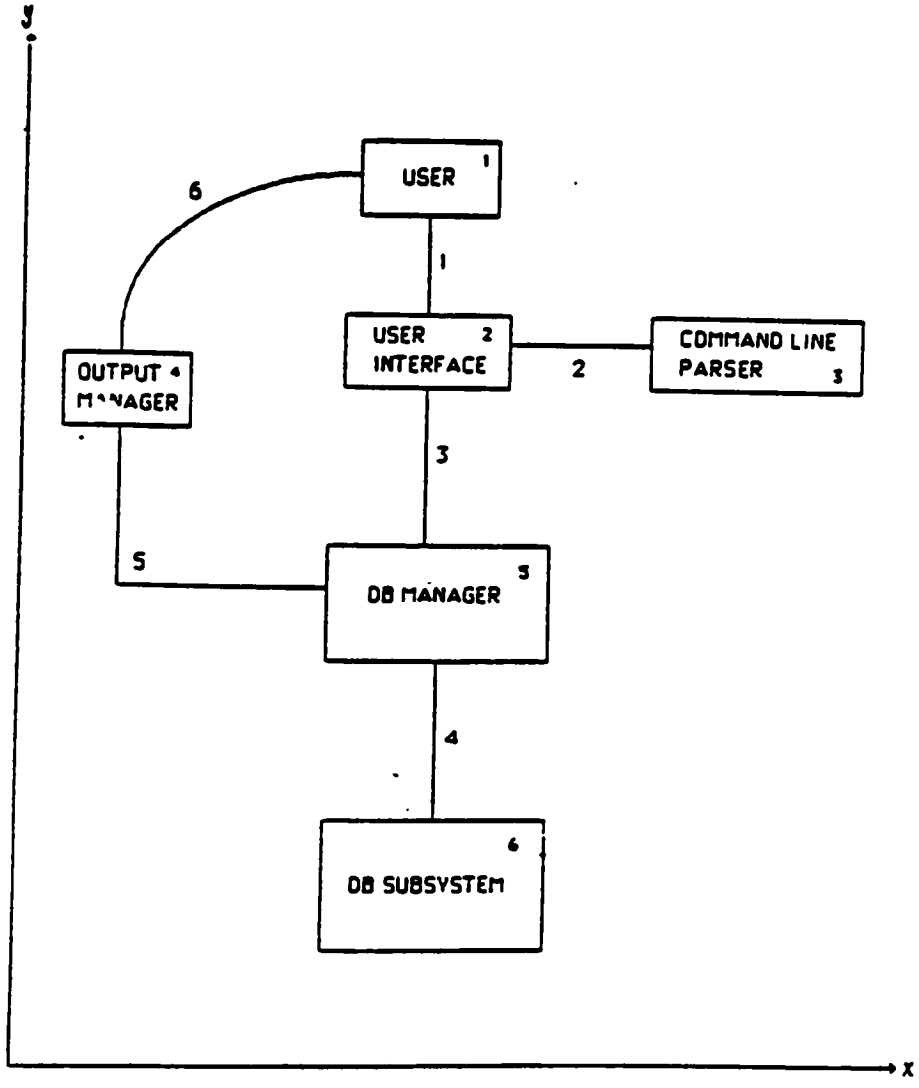

Figure 5.1. Graphical Representation of DBMS

defining an ADT on another previously defined ADT.

Using the above Abstract Data Types defined, the following relations are defined.

```
definerel modules
        mod_id                  int
```

```
        box             box
        level           int
definerel connections
        connection_id   int
        start           cord
        end             cord
        level           int
        toparams        char    63
        fromparams      char    63
definerel linesegments
        connection_id   int
        start           cord
        end             cord
        linetype        int
        thickness       float
```

The following instructions to the DBMS reflect the representation of figure 5.1

```
append modules 1 { {user} {13 26} 4 2 {0.01 1} } 1
append modules 2 { {user interface} {13 20} 6 2 {0.01 1} } 1
append modules 3 { {command line parser} {22 20} 6 2 {0.01 1} } 1
append modules 4 { {output manager} {3 19} 4 2 {0.01 1} } 1
append modules 5 { {db manager} {11 12} 8 4 {0.01 1} } 1
append modules 6 { {db subsystem} {11 3} 8 4 {0.01 1} } 1
append modules 7 { {from db manager} {13.5 6.25} 1 0.5 {0.01 1} } 2
append modules 8 { {relation package} {11.5 4} 2.5 1.5 {0.01 1} } 2
append modules 9 { {record package} {14.5 4} 2.5 1.5 {0.01 1} } 2

append connections 1 {15 24} {15 22} 1 {commands} {error messages}
append connections 2 {18 21} {22 21} 1 {string} {tokens}
append connections 3 {15 20} {15 16} 1 {command arguments} {error messages}
append connections 4 {15 12} {15 7} 1 {system buffers}
        {system buffers; error messages}
append connections 5 {11 14} {5 19} 1 {raw data} {}
append connections 6 {5 21} {13 27} 1 {formated data} {}
append connections 7 {14 6.25} {12.75 5.5} 2 {relation buffers}
        {relation buffers; error messages}
append connections 8 {14 6.25} {15.75 5.5} 2 {record buffers}
        {record buffers; error messages}

append linesegments 1 {15 24} {15 22} 1 0.01
append linesegments 2 {18 21} {22 21} 1 0.01
append linesegments 3 {15 20} {15 16} 1 0.01
append linesegments 4 {15 12} {15 7} 1 0.01
append linesegments 5 {11 14} {5 14} 2 0.01
append linesegments 5 {5 14} {5 19} 2 0.01
append linesegments 6 {5 21} {13 27} 2 0.01
append linesegments 7 {14 6.25} {12.75 5.5} 1 0.01
append linesegments 8 {14 6.25} {15.75 5.5} 1 0.01
```

The **"MODULES"** relation stores the information about each module. Each module is characterized by a *BOX* on the two dimensional plane. The properties of each *BOX* include its location (LL stands for the coordinate of the lower left corner of the *BOX*), its size (*WIDTH* and *HEIGHT*), and its graphical display properties. The inclusion of the *LEVEL* attribute allow storage and retrieval of data at different levels of detail. For example, modules with ID 7, 8, and 9 are actually the detail of module with ID 6 (not shown in figure 5.1). Details of a particular module are located inside the *BOX* that represents the module on the graphics plane. One way of interpreting this is that of zooming into a *BOX* to look at what is stored inside.

Notice that each "*connection*" is given a unique ID (*connection_id*) and can consist of more than one line segment. The line segments that make up one "*connection*" is stored in the **"LINESEGMENTS"** relation. For example, the "*connection*" with an ID of 5 is actually made up of two straight lines.

5.3. Retrieval and Manipulation of Data

Simple retrieval of data can be achieved by simple queries such as

"retrieve box.LL from modules where {level = const int 1}"

which will display the coordinate of the lower left corner of each *BOX* that represents each module. However, in order to effect more powerful data retrieval and manipulation, the user will have to define some operators that act on the ADT's previously defined.

The following examples are definitions of new user-defined operators:

The command

defineop mkvec 3 2 float float vector ../mkvec

defines a new operator called "*mkvec*" which takes two floating point numbers and forms a vector. The precedence of the new operator is three and it is a binary operator. The defini-

tion of the actual sequence of operations for this new operator is to be found in the file with pathname "../mkvec". The content of the file is as follows:

```
RESULT.X_INC = LEFT
RESULT.Y_INC = RIGHT
```

The command

defineop vec_add 3 2 cord vector cord ../vec_add

defines a new operator to perform vector addition on a coordinate using the right operand which is a vector. The file "../vec_add" contains,

```
RESULT.X = {LEFT.X + RIGHT.X_INC}
RESULT.Y = {LEFT.Y + RIGHT.Y_INC}
```

The command

defineop mkcord 3 2 float float cord ../mkcord

defines a new operator which takes two floating point numbers and produces a coordinate type result. The file "../mkcord" contains,

```
RESULT.X = LEFT
RESULT.Y = RIGHT
```

The command

defineop linelength 3 2 cord cord float ../linelength

defines a new operator which takes two coordinates, which are the starting and ending location of a line segment and calculates the length of that line segment. The file "../linelength" contains,

```
%0 = { ( ABS ( RIGHT.X - LEFT.X ) ) ** CONST FLOAT 2.0 }
%1 = { ( ABS ( RIGHT.Y - LEFT.Y ) ) ** CONST FLOAT 2.0 }
RESULT = { ( %0 + %1 ) ** CONST FLOAT 0.5 }
```

The command

defineop pt_on_line 3 2 cord line int ../pt_on_line

defines a slightly more complicated operator which determines if the left operand (a coordinate) actually lies on top of the right operand (a line). The file "../pt_on_line" contains,

```
%0 = { ( RIGHT.END.Y - LEFT.Y ) * ( LEFT.X - RIGHT.START.X ) }
%1 = { ( RIGHT.END.X - LEFT.X ) * ( LEFT.Y - RIGHT.START.Y ) }
%2 = { ( LEFT.X - RIGHT.START.X ) + ( RIGHT.END.X - LEFT.X )
       = ( RIGHT.END.X - RIGHT.START.X ) }
%3 = { ( LEFT.Y - RIGHT.START.Y ) + ( RIGHT.END.Y - LEFT.Y )
       = ( RIGHT.END.Y - RIGHT.START.Y ) }
RESULT = { ( %0 = %1 ) & %2 & %3 }
```

The command

defineop pt_touches_box 3 2 cord box int ../pt_touches_box

defines a new operator which determines if a point on the two-dimensional plane touches the boundary of a *BOX* or not. The definition of the operation is

```
%0 = { RIGHT.WIDTH MK_VEC CONST FLOAT 0 }
%1 = { CONST FLOAT 0 MK_VEC RIGHT.HEIGHT }
%2 = { RIGHT.WIDTH MK_VEC RIGHT.HEIGHT }
%3 = { RIGHT.LL VEC_ADD %0 }
%4 = { RIGHT.LL VEC_ADD %1 }
%5 = { RIGHT.LL VEC_ADD %2 }
%6 = { RIGHT.LL MKLINE %3 }
%7 = { RIGHT.LL MKLINE %4 }
%8 = { %4 MKLINE %5 }
%9 = { %3 MKLINE %5 }
RESULT = { ( LEFT PT_ON_LINE %6 ) |
         ( LEFT PT_ON_LINE %7 ) |
         ( LEFT PT_ON_LINE %8 ) |
         ( LEFT PT_ON_LINE %9 )
         }
```

%0 to %2 are used to calculate the vector that will transform the lower left corner of the box to the other corners of the box. Since the coordinate of the lower left corner of the box is already known, the coordinates of the rest of the three corners are calculated in %3 - %5. Then the four lines that make up the box are calculated in %6 - %9. Finally, the operator

PT_ON_LINE is called to determine if the point touches any one of the lines that make up the boundary of the box.

The command

defineop pt_inside_box 3 2 cord box int ../pt_inside_box

defines an operator to test if a point lies inside a box. The file "../pt_inside_box" contains,

```
%0 = { ( LEFT.X < ( RIGHT.LL.X + ( RIGHT.WIDTH ) ) }
%1 = { ( LEFT.X > ( RIGHT.LL.X ) ) }
%2 = { ( LEFT.Y < ( RIGHT.LL.Y + ( RIGHT.HEIGHT ) ) }
%3 = { ( LEFT.Y < ( RIGHT.LL.Y ) ) }
RESULT = { %0 & %1 & %2 & %3 }
```

The command

defineop box_inside_box 3 2 box box int ../box_inside_box

defines an operator to determine if a box is inside another box or not. The file "../box_inside_box" contains,

```
%0 = { ( RIGHT.LL.X + RIGHT.WIDTH ) }
%1 = { ( RIGHT.LL.X ) }
%2 = { ( RIGHT.LL.Y + RIGHT.HEIGHT ) }
%3 = { ( RIGHT.LL.Y ) }
%4 = { ( LEFT.LL.X + LEFT.WIDTH ) }
%5 = { ( LEFT.LL.X ) }
%6 = { ( LEFT.LL.Y + LEFT.HEIGHT ) }
%7 = { ( LEFT.LL.Y ) }
RESULT = { ( %4 < %0 ) & ( %5 > %1 ) & ( %6 < %2 ) & ( %7 > %3 ) }
```

5.4. Examples of Use of DBMS

With the ADT's and the operators defined in the data base, we can now make more complex inquiries about the current state of the data base. For example, the following command can be used to capture the result of the following retrieval operation into the variable called "$box".

setvar in modules $box = box

where {box.title = const char {db subsystem}}

The variable "$box" can then be used as part of any expression. The following command retrieves all the modules that are defined inside the "DB Subsystem" (at the next lower level).

retrieve * from modules where {box box_inside_box $box}

and the following command will find all the connections inside the box that defines the "DB Subsystem".

retrieve * from connections where {start pt_inside_box $box}

A new relation called "DBSUBSYSCONS" that stores all the connections defined inside the "DB Subsystem" can be created on the fly by the following "view" operation.

```
view {
        { con_id = connection_id }
        { line = {start mkline end} }
} from connections
where { ( start pt_inside_box $box ) }
into dbsubsyscons
```

CHAPTER 6

CONCLUSIONS AND FUTURE ENHANCEMENTS

6.1. Conclusions from The Research DBMS

This thesis is targeted at experimenting with the support of ADT and their associated operators within a relational DBMS. Therefore, other areas of weakness inherent in a relational DBMS with respect to engineering environments, mentioned in section 1.3 will not be discussed here. The following are conclusions drawn by the author from research done on the DBMS developed.

6.1.1. Research DBMS

When designing and implementing a data base management system, the designer is faced with the need to work at a low level, that is, in terms of bytes and bits of raw data. Therefore, to those attempting similar designs, word alignment in manipulating data at such a low level should be handled with extreme care. Much time was spent tracking down program failures that occurred intermittently and most occurred because of word misalignment.

In order to enhance its readability, maintainability and modularity, the DBMS was broken down into three distinct levels -- the user interface, the DB manager, and the DB subsystem. The user interface controls the communication with the user or data base designer. The DB manager is the center of control for both data and control flow. The DB subsystem is the lowest level interface to the UNIX• file system. Global data structures were designed to act as agents for transportation of pertinent data throughout the different levels of the DBMS.

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

This helps in clarifying the design and specification of the different modules within the entire DBMS. It also helps in future modification and enhancement of the DBMS.

Indexing on primary keys of relations, which was originally supported in the simple relational DBMS, is not supported in the research DBMS in order to simplify its design and implementation. This results in a degradation in performance since all queries are handled with sequential a search through the tuples in the relations. Possible improvement in this area is discussed in section 6.2.1

The support of the "record" data type is definitely a deviation from the relational model because data pointers are used to chain up data records to form linked-lists under certain circumstances. The only reason for this implementation is to enhance the performance of the DBMS and these data structures are only used internally by the DBMS. The fact that the data base designer or user is not allowed direct access to these "record" data structures, indicates that the use of these pointers is transparent to the user. The user still views the DBMS as relational.

6.1.2. Support for Abstract Data Types

One important restriction to the definition of ADT in this DBMS is that every element of each ADT being defined has to be either a predefined data type or another previously user-defined ADT. This implies that cyclic definition of ADT, having any element type identical to the data type being defined, is disallowed. Therefore, heterogeneous relations (with tuples having variable lengths) is not supported.

Another restriction imposed upon the data base designer is that all constants used within expressions on the command line have to be preceded by the keyword "CONST" and the data type name of that constant. This is done to simplify the parsing process of expressions. The restriction is not really necessary because the expression parser should be intelligent enough to deduce the actual data type of a constant in an expression by the associated variables and operators.

Users are allowed to define new operators associated with previously defined ADTs by putting instructions in a file which describes, in detail, the exact steps which should be followed to perform that operation on the operands, using pre-defined operators or previously defined operators. With this implementation, the data base designer is not restricted by any particular programming language required to implement the operator.

Instructions for performing an operation are parsed and transformed into post-fix expressions for storage and future usage. Inevitably, this introduces a certain amount of overhead in retrieving these stored post-fix expressions for evaluation, as compared to the alternative of compiling and linking user-written functions that implement the same operation. In order to improve the performance of the DBMS with respect to the evaluation of complex expressions, a cache mechanism is used to keep the definition of previously used operators in virtual memory. This enhanced the performance of the DBMS greatly when complex expressions have to be evaluated for each tuple being searched.

6.1.3. Support for Design Data Version Control

The use of "version words" to control the visibility of tuples in the data base significantly reduces duplication of the amount of data needed to be stored. One drawback of this design is that the maximum number of alternatives, and versions in each alternative, has to be specified at the time a data base is initiated.

All versions in the same alternative, other than the most current version, are considered frozen. This means that older versions are not allowed any alterations at all. Designs being worked on simultaneously should be kept in the data base as different alternatives.

6.2. Suggested Future Enhancements to the DBMS

6.2.1. Indexing for Efficiency

The ability of the data base designer to specify index files on primary keys in a relation will greatly improve the retrieval of data from the data base. However, the support for ADT

provided in this DBMS poses a problem with supporting indexing when the candidate key being indexed happens to have a user-defined ADT as its data type. Basically, the user has to define an operator, (or a set of operators), which the DBMS can invoke to construct an ordered index file on the ADT. Retrieval procedures will have to be informed of the specific operators needed to be invoked to index into the index file using the primary key.

6.2.2. Programming Interface for DBMS

The research DBMS is written to be command line driven and is basically interactive. In order to support more user friendly applications built on top of this DBMS, a programming interface to the DBMS is needed. Preferably, this interface should also support access of the data managed by the DBMS over the network. This will require that a DBMS server be started in the node where the actual data base resides. This server will listen to a well-known address for requests to access the data bases under its control.

This interface will be at the same level as the user interface, but instead of reading command lines from a terminal, it will receive its directions from a network socket. This makes available to users in a local area network the middle level functions managed by the DB manager.

6.2.3. Compilation of ADT Operators

The support for defining ADT operators implemented in the research DBMS is excellent for interactive use, testing and debugging of new ADT operators. However, once an operator has been tested, it would be advantageous if the actual code can be compiled and linked into the DBMS itself to accelerate the execution of the operator. However, there is no provision under the UNIX• operating system for dynamic linking of modules into a running process. Consequently, a special linker needs to be developed for this purpose.

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

APPENDIX A

APPENDIX A - SYSTEM BUFFERS

The following are the system-wide data buffer definitions used for passing information and data between different modules in the DBMS.

A _RELATIONBUF is used to hold information about a relation file.

```
typedef struct {
        char rel_name[R_NAME_LEN];   /* name of relation              */
        char rel_fname[F_NAME_LEN];  /* file name of relation         */
        int  rel_fd;                 /* file descriptor of file       */
        int  curindex;               /* index to current tuple        */
        int  next_free;              /* next free block in free list  */
        int  empty_records;          /* no of free blocks in file     */
        int  tuplesize;              /* size (in bytes) of 1 tuple    */
        char *data;                  /* space for data transfer       */
} _RELATIONBUF;
```

A _RECORDBUF is used to store information about a record file.

```
typedef struct {
        char rec_fname[F_NAME_LEN];  /* file name of record file      */
        int  rec_fd;                 /* file descriptor of file       */
        int  rec_no;                 /* start index of record         */
        int  tblk;                   /* total blocks in record        */
        int  nblk;                   /* no of blocks already read in  */
        int  next_free;              /* next free block available     */
        int  empty_records;          /* no of free blocks in file     */
        int  curblk;                 /* index to current block        */
        int  blocksize;              /* size of one data block        */
        char *data;                  /* space for data transfer       */
} _RECORDBUF;
```

An array of _ATTBUF is used to store the information of a list of attributes. This includes their names, data types and offsets.

```
typedef struct {
        char att_name[A_NAME_LEN];   /* name of attribute          */
        int  att_type;               /* data type of attribute     */
        int  att_offset;             /* offset in tuple data space */
} _ATTBUF;
```

An _OPBUF is used to store information about the characteristics of an operator.

```
typedef struct {
        char opname[O_NAME_LEN];     /* name of operator           */
        int  type;                   /* predefined or ADT          */
        int  opindex;                /* index to ADTOPS            */
        int  card;                   /* cardinality of operator    */
        int  lopnd;                  /* left operand data type     */
        int  ropnd;                  /* right operand data type    */
        int  precedence;             /* precedence of operator     */
        int  resulttype;             /* result data type           */
} _OPBUF;
```

An array of _EXPRBUF is used to store a parsed expression in post-fix format for storage and evaluation.

```
typedef struct {
        char token[TOKEN_LEN];       /* character form of token    */
        int  ident;                  /* token class                */
        char *addr;                  /* actual address             */
        int  offset;                 /* byte offset in work space  */
        int  type;                   /* data type of token         */
        _OPBUF   opbuf;              /* if it's an operator        */
} _EXPRBUF;
```

An _ASSIGNSTAT buffer stores a parsed assignment statement for later retrieval and execution.

```
typedef struct {
        int          destoffset;             /* offset in workspace for result   */
```

```
        int          nexpr;                    /* no of elements in parsed expression */
        _EXPRBUF     expr[MAXEXPR];            /* actual stored expression            */
} _ASSIGNSTAT;
```

A _VARBUF holds information about a user-defined variable.

```
typedef struct {
        char var_name[80];          /* name of variable                    */
        int  var_type;              /* data type of variable               */
        char *var_addr;             /* location of actual value of variable */
        int  var_size;              /* internal size of variable           */
} _VARBUF;
```

An array of _OPSTAT is used to store a sequence of assignment statements that performs an user-defined operation on ADTS.

```
typedef struct OPSTAT {
        _ASSIGNSTAT    assignstat;      /* assignment statement      */
        struct OPSTAT  *next;           /* next assignment statement */
} _OPSTAT;
```

A system maintained array of _OPCACHE is used to store the information about user-defined operators in core memory to speed up the evaluation process of expressions.

```
typedef struct {
        char opname[O_NAME_LEN];    /* operator name                    */
        int  usecnt;                /* usage frequency count            */
        _ATTBUF  *opatts;           /* attributes involved              */
        char     *workspace;                /* actual work space allocated */
        _OPSTAT  *opstatement;      /* list of assignment statements*/
} _OPCACHE;
```

APPENDIX B

APPENDIX B - PROGRAM LAYOUT

---

adt.c

---

defineadt(field) char *field[];

Define an ADT using info in command line. Update ADTS & ADTMEMS.

findadt(adtname, adtid, adtmems, totalsize)

char *adtname;
_ATTBUF *adtmems;
int *adtid;
int *totalsize;

If ADT with adtname is defined, info about that ADT is placed in adtid, totalsize, and its elements & their types are placed adtmems. Returns the no of members defined for that ADT.

identadt(adtid, adtmems, totalsize)

int adtid;
_ATTBUF *adtmems;
int *totalsize;

Identifies the ADT with adtid as its ID and fills in the adtmems array and totalsize (size of internal representation of ADT). Returns the number of elements in the ADT.

adtsizeof(adtid) int adtid;

Returns the no. of bytes it takes to store the ADT internally.

tsizeof(type) int type;

Returns the internal size of the type (basic or ADT).

typename(type, name) int type; char *name;

Returns the character name of the "type" in "name".

findtype(name) char *name;

Returns the internal enumerated ID of type by the name "name". Returns -1 if type "name" is not defined.

---

adtops.c

---

padtop(opbuf, lopndaddr, ropndaddr, resultaddr)

_OPBUF *opbuf;
char *lopndaddr, *ropndaddr, *resultaddr;

Performs an operation defined on ADT's. Memory space is malloced to results of operations. Actual addresses are bond to relative offsets of variables calculated when the expressions were parsed. Then *evalexpr* is called to evaluated the expression upon the data in the workspace formed with the operands and result. Finally, the result is placed in location addressed by "resultaddr".

defineop(nfields, field) int nfields; char *field[];

Reads the statements that define the operation of a new operator on ADT's.
Statements are parsed and stored for future use by *padtop*.

static checksyntax(nfields, field, opbuf, defn_fd)

Check for the correct syntax of a "defineop" command. The syntax is
DEFINEOP <adtopname> <precedence> <card> <ltype> <rtype>
<resulttype> <defn filename>
Returns -1 if fail.

static build_latts(nlatts, latts, ltype, rtype, resulttype)

```
int             nlatts;
_ATTBUF         latts[];
int             ltype, rtype, resulttype;
```

Build up local attribute list.

static cleanup()

Perform cleanup upon error detected.

---

append.c

---

append(nfields, field) int nfields; char *field[];

Append a new tuple to a relation.

---

definerel.c

---

definerel(nfields, field) int nfields; char *field[];

Define a new relation in design data base.

---

delete.c

---

delete(nfields, field) int nfields; char *field[];

Delete tuples from a relation or delete the entire relation.

---

display.c

---

display(nfields, field) int nfields; char *field[];

General function to display data in data base.

static show_rel_tbl() char tname[A_NAME_LEN]; int relcnt = 0;

Display information about relations defined in data base.

static show_adts()

Display information concerning ADT's defined by user.

static show_ops()

Display information about user defined operators.

static show_vers()

Display information about defined versions in data base.

---

expr.c

---

parseexpr(field, expr, nrelatts, relatts, resulttype)

```
char            *field[];
_EXPRBUF        expr[];
```

```
        int             nrelatts;
        _ATTBUF         relatts[];
        int             *resulttype;
```
Parse the expression whose elements are in field, using the info about the relation as given in nrelatts and relatts. The result is a postfix expression in expr which is ready to be evaluated. Returns the number of elements in the postfix expression built.

evalexpr(expr, nelements, data, result)
```
        _EXPRBUF    expr[];
        int         nelements;
        char        *data;
        char        *result;
```
Evaluate the expression in expr with nelements elements, the actual data being used in the evaluation is given in data and the result is put into the location addressed by result.

caloffset(name, type, natts, atts) char *name; int *type; int natts; _ATTBUF atts[];
Identify the real type of the identifier "name" and calculates its offset from the starting addr of a tuple in that relation.

static initopstack()
Initialize the operator stack to empty.

static opstackempty()
Determines it the operator stack is empty (1) or not (0).

static pushopstack(opbuf) _OPBUF *opbuf;
Pushes the operator buffer opbuf onto the operator stack.

static popopstack(opbuf) _OPBUF *opbuf;
Pop the operator buffer on the top of operator stack & copy it to opbuf.

static initpstack()
Initialize the parsing/process stack to empty.

static pstackempty()
Determines if the parsing/process stack is empty (1) or not (0).

static pushpstack(element) int element;
Push element (index to expr) onto top of parsing/process stack.

static poppstack(element) int *element;
Pop the element on top of parsing/process stack and copy it to element.

---

main.c

---

main(argc, argv) int argc; char *argv[];
Main function of DBMS.

getaline(inbuf, infileptr, echo) char *inbuf; FILE *infileptr; int echo;
Get one line of input from infileptr. The "echo" option tells it whether to echo the command or not.

intr()
Interrupt handling routine.

static initialize()
Initialization of data base

static restart()
Restart an already initialized data base

static setversion()
Prints out the description of all the currently defined versions and ask user which one he wants.

static helpusr()

Prints out the help manual

---

op.c

---

is_op(token, opbuf) char *token; _OPBUF *opbuf;
If token represents an operator, gather its info in to opbuf else return -1.

ckopnds(opbuf, ltype, rtype, resulttype)
_OPBUF *opbuf;
int *ltype;
int *rtype;
int *resulttype;
Check if the left right opnd types are correct and set the resulttype
If either of the opnd is a constant set its type to what it should be
Only allow one constant for binary ops and none for unary ops.

performop(opbuf, lopndaddr, ropndaddr, opndtype, result)
_OPBUF *opbuf;
char *lopndaddr;
char *ropndaddr;
int opndtype;
char *result;
Perform the operation specified in opbuf upon the given operands and put back the result in location addressed by result.

static pbaseop(opbuf, lopndaddr, ropndaddr, resultaddr, type)
_OPBUF *opbuf;
int lopndaddr, ropndaddr, resultaddr, type;
Perform a basic operation on the operands.

static loadop(opname, ADTOPSindex) char *opname; int ADTOPSindex;
Load the definition of an ADT into the OPCACHE.

---

output.c

---

initoutq(natts, atts) int natts; _ATTBUF atts[];
Initialize the output queue to null. natts & atts are used to tell the following functions how to interpret and output the data. These have to be set by the caller to determine which parts of the tuples are to be printed.

queueoutq(data) char *data;
Queue the parts of the data that are to be printed to the output queue. Data is transformed from internal representation to external representation.

flushoutq()
Flush everything in the output queue and line them up in columns. After that, cleanup the output queue next display.

static outqempty()
Test if the output queue is empty or not.

static cleanoutq()
Clean the output queue for the next caller.

static printrec(rec) char * rec;
Print out a single record in the output queue using the format built up previously.

static printheader()
Print the header for each output page to show what each column means.

---

parse.c

---

parse(buffer) char buffer[];
Parse the input buffer and find out what the user wants.
breakup(line, field) char *line; char **field;
Break up the line in fields. The space is the normal field delimiter. Sublists are enclosed within {}. Each breakup strips off one matching pair of {}.

---

project.c

---

project(nfields, field) int nfields; char *field[];
Project a relation over certain attributes.

---

record.c

---

initrecfile(rec_fname) char *rec_fname;
openrecfile(record) _RECORDBUF *record;
Opens the record file and put fd in record->rec_fd.
closerecfile(record) _RECORDBUF *record;
Closes the record file and sets record->rec_fd to NIL (-1).
newrecord(record) _RECORDBUF *record;
Sets record->rec_no to NIL.
addblock(record) _RECORDBUF *record;
Appends block in record->data to record->rec_no (set when the first block was added).
openrecord(record) _RECORDBUF *record;
Finds the record pointed to by record->rec_no, sets tblk to total no. of blocks for this record, sets nblk to zero to begin with, ans sets curblk to the first block in the record.
getnextblock(record) _RECORDBUF*record;
Get the block pointed to by record->curblk and put it in record->data
Updates record->nblk to reflect how many blocks have been read
Returns -1 if no more blocks to read.
delrecord(record) _RECORDBUF *record;
Free the record pointed to by record->rec_no to the free list and resets record->rec_no to NIL (-1).

---

relation.c

---

initrelfile(rel_fname) char *rel_fname;
openrelfile(R) _RELATIONBUF *R;
closerelfile(R) _RELATIONBUF *R;
addtuple(R) _RELATIONBUF *R;
Add tuple in R->data to relation file R->rel_fname
Expects file to be already opened.

deltuple(R) _RELATIONBUF *R;
Delete the tuple specified in Rbuf R from relation (set by previous find operations).
updatetuple(R) _RELATIONBUF *R;
Update a tuple (found from previous find operation and then modified).
rewindrel(R) _RELATIONBUF *R;
Starts the search from front again.
getnexttuple(R) _RELATIONBUF *R;
Get the next tuple in relation file into data & updates curindex
Returns (-1) if no more is available.
newalt(R, newalt) _RELATIONBUF *R; int newalt;
Dups the currently visible tuples in the relation in newalt
Must be called before actually setting curalt to new value.

---

services.c

---

itoa(n, s) char s[]; int n;
find_rel(relation, atts) _RELATIONBUF *relation; _ATTBUF *atts;
If relation is defined, fills in the atts array for its attributes and return the no. of attributes defined; else return -1.
gettime(cur_time) char *cur_time;
Get the current time and date and format it to a 14 char string.
bcopy(source, dest, nbytes) char *source, *dest; int nbytes;
Copy nbytes from source to dest.
is_int(charptr) char *charptr;
Check if string in charptr represents an int or not. Returns -1 if not int, 0 if int.
is_float(charptr) char *charptr;
Check if string in charptr represents an float or not. Returns -1 if not float, 0 if float.
fold(ptr) char *ptr;
Folds string in ptr to all upper case.
datacmp(addr, type, key) char *addr; int type; char *key;
Compare data in addr to that in key, interpreting them as type.
Returns 0 if equal; 1 if key > addr; -1 if key < addr.
copyrelbuf(src, dest) _RELATIONBUF *src, *dest;
Copies _RELATIONBUF structure src to dest.
copyrecbuf(src, dest) _RECORDBUF *src, *dest;
Copies _RECORDBUF structure src to dest.
copyopbuf(src, dest) _OPBUF *src, *dest;
Copies _OPBUF structure src to dest.
copyattbuf(src, dest) _ATTBUF *src, *dest;
Copies _ATTBUF structure src to dest.
buildattbufs(field, attmems, totalsize)
char *field[]; _ATTBUF *attmems, int *totalsize;
Decipher the attributes and their types specified in field and put them into attmems array. Returns the number of attributes built. Also return totalsize of all attributes.
storeattbufs(natts, attmems, C) int natts; _ATTBUF *attmems; _RECORDBUF C;
Stores array attmems away in record file C. Returns rec_no storing it.
expo(x, y) double x, y;

Returns the value of x to the power y.

int_to_ext(src, dest, type) char *src, *dest; int type

Convert internal representation of data of "type" in src into its external representation in dest.

ext_to_int(src, dest, type) char *src, *dest; int type

Convert external representation of data of "type" in src into its external representation in dest. Checks for validity of the src string in representing a value of "type". Returns -1 if not valid.

pad(no_of_sp, padchar) int no_of_sp; char padchar;

Fill "no_of_sp" character positions with "padchar" characters

---

update.c

---

update(nfields, field) int nfields; char *field[];

Perform an update operation.

---

var.c

---

is_var(name, varbuf) char*name; _VARBUF*varbuf;

Determine if the name represents a user-defined variable or not. If it is, put its info into varbuf.

setvar(nfields, field) int nfields; char *field[];

Set a variable to a value of a constant or the result of a retrieval operation on a relation for later use.

showvar(nfields, field) int nfields; char *field[];

Show the value of a user-defined variable.

unsetvar(field) char *field[];

Free the memory space used to store a variable.

---

version.c

---

chalt(new) char *new;

Switch current context to another previously defined alternative.

newalt()

Create a new alternative of the design in data base based upon the data visible in the current alternative and version.

chversion(new) char *new;

Switch the context to another previously defined version of the design in data base

newversion()

Freeze the current state of design and start working with a new version

REFERENCES

[ATWO85] Thomas M. Atwood, "An Object-oriented DBMS for Design Support Applications", Compint 1985 IEEE CH2136-0 pp. 299 - 307.

[BANE85] Won Kim, Jay Banerjee, "Support of Abstract Data Types in a CAD Database", Compint 1985 IEEE CH2136-0 pp. 381 - 385.

[BARA85] G. P. Barabino, G. S. Barabino, G. Bisio, M. Marchesi, "Improved Data Management Performances of an Integrated System of CAD Tools.", Compint 1985 IEEE CH2136-0 pp. 401 - 405.

[BUCH84] Alejandro P. Buchmann, "Current Trends in CAD Databases", Design Automation, vol 16 #3 may 1984 Butterworth & Co (Publishers) Ltd.

[GADI85] A. J. Gadient, "Functional Requirements for An Electronic Design Automation Environment Integration Framework", Compint 1985 IEEE CH2136-0 pp. 348 -354.

[HARD84] Martin Hardwick, "Extending the Relational Database Model for Design Applications", 1984 21st Design Automation Conference, IEEE, Paper 8.2

[HAYN83] Mark N. Haynie, "Tutorial: The Relational Data Model for Design Automation", 1983 20th Design Automation Conference, IEEE, Paper 38.2

[JAIN85] Hemant K. Jain, Dimitrios A. Liaskos, "Advances in Distributed DBMS for CAD/CAM", University of Wisconsin-Milwaukee, Compint 1985 IEEE CH2136-0 pp. 368 - 370.

[JOHN85] H. Randall Johnson, "CAD/CAM Data Management Research on the EMIS Project", Boeing Computer Services, Seattle, Washington, (Compint 1985 IEEE CH2136-0 pp. 371 - 380).

[KIM84] W. Kim, R. Lorie, D. McNabb, W. Plouffe, "A Transaction Mechanism for Engineering Design Databases.", Proceedings of 10th International Conference on Very Large Data Bases (1984) pp. 355 - 362.

[KORE75] A. J. Korenjak, A. H. Teger, "An Integrated CAD Data Base System", Proceedings of 12th Design Automation Conference, 1975, pp. 399 - 406.

[LENZ84] Han Diel Gerald Kreissig Norbert Lenz, Michael Scheible Bernd Schoener, "Data Management Facilities of an Operating System Kernel", ACM vol 6, August 1984, pp. 58 - 69.

[LOCK79] Lockman, P. et al. "Data Abstraction for Data Base Systems." TODS, 4, 1, March 1979.

[LOOM85] Mary E. S. Loomis, "Logical Data Modeling - A Step Towards Integration.", Compint 1985 IEEE CH2136-0 pp. 392 - 400.

[LORI83] Raymond Lorie, Wilfred Plouffe, "complex Objects and Their Use in Design Transactions", 1983 IEEE Proceedings of Annual Meeting - Database Week, pp. 115 - 121.

[LORI85] Klaus R. Dittrich, Raymond A. Lorie, "Object-oriented Database Concepts for Engineering Applications", Compint 1985 IEEE CH2136-0 pp. 321 - 325.

[MASA74] Masakazu, Soga, et al, "Engineering Data Management System (EDMS) for Computer Aided Design of Digital Computers", Proceedings of 11th Design Automation Conference, 1974, pp. 372 - 279.

[ROWE79] Rowe, L. and Schoens, K., "Data Abstraction, Views and Updates in RIGEL." Proc 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass. May 1979.

[SCHM78] Schmidt, J., "Type Concepts for Database Definition." Proc. International Conference on Data Bases, Haifa, Israel, August 1978.

[SIDL80] Thomas W. Sidle, "Weaknesses of Commercial Data Base Management Systems in Engineering Applications", Communications of the ACM (June 1980), pp. 57-61

[STON83] Michael Stonebraker, Brad Rubenstein, Antonin Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Data Bases." IEEE 1983 Proceedings of Annual Meeting - Database Week.

[STON82] Stonebraker, M. R., Kalash, J., "Timer: A Sophisticated Relation Browers.", Proceedings of 8th International Conference on Very Large Data Bases, 1982, pp. 1 - 10.

[VERN85] F. Vernadat, W. H. Henneker, "CAD/CAM Databases at NRC: From Manufacturing cell Database Systems to Engineering Information Systems", Compint 1985 IEEE, CH2136-0 pp. 441 - 446

[VU85] Khanh Vu, "CORD Data System.", 1985 CADTEC Corporation, San Jose, CA

[WASS79] Wasserman, A. I., "The Data Management Facilities of PLAIN." Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., May 1979.

[WONG79] S. Wong, W. A. Bristol, "A Computer Aided Design Data Base", Proceedings of 16th Design Automation Conference, 1979, pp. 398 - 402.