The Implementation of a Network Oriented Database Management System Designed to Run Under the Unix Operating System

Mary Dvonch

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation
THE IMPLEMENTATION OF A NETWORK ORIENTED
DATABASE MANAGEMENT SYSTEM

DESIGNED TO RUN UNDER THE UNIX* OPERATING SYSTEM

A thesis submitted in partial fulfillment of the
Master of Science in Computer Science Degree Program

by

MARY ANN DVONCH

Approved by: Peter H. Lutz
Peter Lutz - Committee Chairman

William Stratton
William Stratton Committee Member

Roy D Czernikowski
Roy Czernikowski Committee Member

Michael J. Lutz
Michael Lutz - Special Technical Advisor

January 1982

*UNIX is a Trademark of Bell Laboratories
Title of Thesis:

THE IMPLEMENTATION OF A NETWORK ORIENTATED DATABASE MANAGEMENT SYSTEM DESIGNED TO RUN UNDER THE UNIX* OPERATING SYSTEM

I, Mary Ann Dvonch, prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

Computer Science Department
Rochester Institute of Technology
One Lomb Memorial Drive
Rochester, New York 14623

*UNIX is a Trademark of Bell Laboratories
I would like to thank my thesis advisor, Dr. Peter Lutz, for the endless hours he spent directing my efforts on this project. Without his help and guidance, this project could not have been completed. I would also like to thank Mr. William Stratton for the ideas for this project and for the use of his B-tree programs. I wish to thank Dr. Roy Czernikowski for the time he has spent as a member of my committee. I would like to thank Mr. Michael Lutz for the time he spent as special technical advisor. Last, but certainly not least, I would like to thank my husband, Jerry, and my children for their help and support during the evolution of this project.
ABSTRACT:

THE IMPLEMENTATION OF A NETWORK ORIENTED DATABASE MANAGEMENT SYSTEM

DESIGNED TO RUN UNDER THE UNIX* OPERATING SYSTEM

We have designed a network approach, low level access system. Our conceptual model is specifically designed to run under the UNIX operating system. Our model has been tooled in the UNIX tradition. We have also supplied a manipulation language for use on the database. Our system does its own input and output buffering.

*UNIX is a Trademark of Bell Laboratories
# TABLE OF CONTENTS

1. **INTRODUCTION**
   1.1. DBTG NETWORK MODEL OVERVIEW 1-2
   1.2. PROPOSED SYSTEM 1-4
   1.3. ASSUMPTIONS AND DESIGN PRIORITIES 1-4

2. **A NETWORK APPROACH DATA MODEL**
   2.1. MEMBERSHIP CLASS 2-1
   2.2. MODEL ENTITIES 2-2
       2.2.1. RECORD TYPES 2-2
       2.2.2. SET TYPES 2-4
   2.3. DATA STRUCTURES EMPLOYED 2-5
       2.3.1. B-TREE INDEXES 2-6
       2.3.2. TABLE STORAGE IN BINARY TREES 2-6
       2.3.3. EXPRESSING SET RELATIONSHIPS 2-8
   2.4. INPUT/OUTPUT BUFFERING 2-9
   2.5. ERROR HANDLING 2-10

3. **PROTOTYPE**
   3.1. RECORD TYPE DESCRIPTIONS AND CONTENTS 3-1
   3.2. SET TYPE DESCRIPTIONS AND CONTENTS 3-3
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. AVAILABLE PRIMITIVES</td>
<td></td>
</tr>
<tr>
<td>4.1. DATABASE DEFINITION DIRECTIVES</td>
<td></td>
</tr>
<tr>
<td>4.1.1. RECORD TYPE ADD</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.2. SET TYPE ADD</td>
<td>4-2</td>
</tr>
<tr>
<td>4.2. DATABASE MANIPULATION PRIMATIVES</td>
<td></td>
</tr>
<tr>
<td>4.2.1. DATABASE EXPANSION</td>
<td></td>
</tr>
<tr>
<td>4.2.1.1. ADD RECORD</td>
<td>4-3</td>
</tr>
<tr>
<td>4.2.1.2. ADD OWNER</td>
<td>4-4</td>
</tr>
<tr>
<td>4.2.1.3. ADD MEMBER</td>
<td>4-4</td>
</tr>
<tr>
<td>4.2.2. DATABASE CONTRACTION</td>
<td></td>
</tr>
<tr>
<td>4.2.2.1. DELETE RECORD</td>
<td>4-5</td>
</tr>
<tr>
<td>4.2.2.2. DELETE MEMBER</td>
<td>4-5</td>
</tr>
<tr>
<td>4.2.2.3. DELETE OWNER</td>
<td>4-6</td>
</tr>
<tr>
<td>4.2.3. DATABASE NAVIGATION</td>
<td></td>
</tr>
<tr>
<td>4.2.3.1. FIND RECORD</td>
<td>4-7</td>
</tr>
<tr>
<td>4.2.3.2. FIND OWNER</td>
<td>4-8</td>
</tr>
<tr>
<td>4.2.3.3. FIND FIRST</td>
<td>4-8</td>
</tr>
<tr>
<td>4.2.3.4. FIND NEXT</td>
<td>4-9</td>
</tr>
<tr>
<td>4.2.4. DATABASE REORGANIZATION</td>
<td></td>
</tr>
<tr>
<td>4.2.4.1. CHANGE OWNER</td>
<td>4-10</td>
</tr>
<tr>
<td>4.2.4.2. CHANGE ALL</td>
<td>4-11</td>
</tr>
<tr>
<td>4.2.5. SAY GOOD-BYE</td>
<td>4-11</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

5. CONCLUSIONS AND FUTURE EXTENSIONS

5.1. IMPLEMENT A SUB-SCHEMA WITH A DATA SUBLANGUAGE 5-1
5.2. TAKING OUT THE TRASH 5-2
5.3. DELETION OF RECORD AND SET TYPES 5-4
5.4. SPIFFY OPTIONS 5-5

BIBLIOGRAPHY

APPENDIX A: LISTING OF DATABASE MANAGEMENT PROGRAMS
1. INTRODUCTION

Out of the ever growing use of computers to store and manipulate data has grown the need to impose some structure on the data, facilitating the use of the data. To answer to those needs, the concepts of database systems and database management have been developed. A database is a collection of stored operational data used by an application system [Date 77]. A database management system provides the user or users with control of this operational data. The heart of any database system is the data model (or conceptual model). Although there are three well known approaches, relational, hierarchical, and network, we have decided to direct our attention to the network approach for our data model.

In a network approach, an entity is represented by a "record" and the association between entities is represented with a "link". The links are used to group records into sets. Each set type may have any number of occurrences. Each set occurrence has one owner record and any number of member records. However, a specific record type may not be an owner and a member in the same set. A network database system can implement many to many mappings more efficiently than a relational system, since a relational system would require more space to represent the mappings. Also, the network model tends to be more symmetric than the hierarchical model. Although a network model may offer more generality than a hierarchical model and may be more efficiently implemented than a relational model, there is little doubt that a network approach is more difficult to conceptualize [Date 77]. A network model requires an understanding of both record types and links, and their interrelationship. The implementation of many-to-many relationships is not straightforward, often requiring dummy record types. With practice one gets used to this technique. While the network approach to databases may be
conceptually more difficult to understand, we have made every effort to ensure simplicity and clarity of usage.

1.1. DBTG NETWORK MODEL OVERVIEW

A network data model is backed by the Conference on Data Systems Language (CODASYL) Data Base Task Group (DBTG). The reports of DBTG in 1971 and 1976 have influenced the national and international standards for databases. We have been guided in the design of our database system by the DBTG reports. The schema basically consists of the implementation of (1) schema declarations (DDL) and (2) a set of primitives for manipulation (DML). The following paragraphs compare our system with the DBTG proposal.

While the DBTG proposal included dividing the total storage space into a number of "areas", our system has only one area in which all stored data resides. We feel this single area is reasonable in our model since we do not envision the use of the present model as a large scale database.

The DBTG proposal allows the user to specify a membership class by means of an INSERTION/RETENTION entry. The INSERTION entry determines the storage class for a record type while the RETENTION parameter determines a removal class [Date 77]. The DBTG proposal allows AUTOMATIC or MANUAL as a storage class and FIXED, MANDATORY, or OPTIONAL for a removal class, with any combination of storage/removal class for any set type. That is, a given record type may have a different membership class in different set types. MANUAL storage requires that the user insert a member into a set occurrence with a specific "insert member" command. With AUTOMATIC storage, the database management system automatically inserts the member into the appropriate set.
FIXED retention dictates that once a record has been entered as a member in a set occurrence, that member can never have any existence in the database not as a member in that set occurrence. MANDATORY retention is like fixed except that a member is allowed to be moved to a new owner within a set type. OPTIONAL retention allows a member to be removed from a set occurrence without completely removing the member from the database. Given the time limitations of this project, we could not begin to support all of these membership classes. Instead, we chose a specific membership class (MANUAL/MANDATORY - See Chapter 2, Section 1) and implemented it across all record and set types. We have made every effort to tool the model in such a way as to allow future extensions to provide a richer set of membership classes.

DBTG allows several forms of set selection. The set selection clause may be thought of as defining an "access strategy" for the set [Date 77]. Our system places the responsibility for selecting the correct set occurrence on the user. He must supply the name of a set he expects to use. However, all of our routines operate through user established currency pointers, so it would be a relatively easy modification to allow the user the option of supplying the set name or using the current of run set type as a default.

The DBTG data model allows for both ACTUAL and VIRTUAL sources for data items. A VIRTUAL data item is one that is not physically stored in a record, but rather could be produced by applying some function to existing fields. Alternatively, the source of a data item could be declared as ACTUAL, which indicates that the data item actually exists in the database. The data item could be a field or a group of fields. Our network model does not deal in fields, other than to produce keys and to make sure that the number of fields in a record matches the
number of fields in its type definition. It would seem reasonable that the software that provides a data sublanguage to the user might be able to provide some virtual capabilities.

1.2. PROPOSED SYSTEM

Figure 1-1 depicts the proposed system. The user/database system interface includes software to provide a subschema definition language and a stand alone user query language. It is hoped that this software will be provided as a future extension of this system. See Section 5-1 for further information and clarification. The Database Management System presently implemented provides definition directives, which allow the creation of record and set types, and manipulation commands or primitives, which allow the manipulation of the data stored in the database. The definition directives are accepted by the DBMS and the appropriate record or set types are defined. The record and set definitions are maintained in a binary tree in memory while the Database Management System is in use. When the DBMS is not being used, the definitions are stored on a file. See Section 2.4.2 for further information. The manipulation primitives work through an input/output buffering system maintained by the DBMS. See Section 2.4 for further information. The manipulation primitives are discussed in detail in Section 4.

1.3. ASSUMPTIONS AND DESIGN PRIORITIES

Our network database model assumes that any record stored in the database has a unique key. Under our present system, "unique key" is defined as a combination of one to ten fields to produce a record identification which is unique throughout the system. Of course the limit of ten fields is merely a parameter and easily changed. It would also be a minor change to require the key to be
DATA & MANAGEMENT SYSTEM

DATA BASE MANAGEMENT FILE

DATA BASE INFORMATION FILES:
1. B-TREE FILES
2. RECORD FILES
3. SET LINK FILES

FIGURE 1-1
PROPOSED SYSTEM
unique only within a record type and not throughout the entire system. Null keys are not allowed, nor is any field that is part of a key allowed to be null. We assume that only one user will be using the database at a time, so as to avoid the simultaneous update problem.

In general, the goal of efficient operation of a database system is bought at the cost of loss of flexibility of design, and flexibility of design causes some loss of efficiency of operation [Inmon 81]. In the trade-offs between performance versus flexibility, it is easy to opt for performance because the utility of performance is very apparent and immediately recognizable. On the other hand, the utility gained by building elasticity into a data structure doesn’t blossom until a later time. Data that is flexible survives best in the face of environmental change. It is only when data that is inflexible undergoes change that the wisdom of building flexible data structures becomes apparent. Along with data flexibility, or perhaps as an extension of it, we have added the goal of ease of extension. Our conception of this system is not that it is a total database package that will serve the needs of all users. Rather, this system is meant to serve as the kernel of a much larger database system. Thus, our efforts were centered on producing a system that would allow maximum flexibility and ease of extension. Obviously, these were paid for at times at the cost of loss of efficiency in either space or time.

A second and very real priority was the optimization of application development time. We attempted to design and implement a well defined system within the given time frame. Given unlimited time and resources, a larger, more powerful system could have been developed. However, it is hoped that the flexibility built into our system will allow it to meet changing user needs and expectations.
Since a network model is at best intricate, we emphasized ease of modification when designing our data structures and the create, delete, and find functions. One of our major concerns was minimizing pointer chasing. C was most helpful in dealing with the pointer problems, and we feel that the pointer design and implementation is one of the strengths of our system. We also attempted to keep the data structures as straightforward as possible in order to ease understanding. Although network models, by their very nature, are more difficult to understand than relational models, we hope that by emphasizing ease of manipulation in the project, we have also added ease of understanding.

Since optimization of execution time and minimizing data storage were not our main concerns, they have undoubtedly suffered. We did supply our own input/output buffering system in an attempt to lower execution time. We stored pointers to data instead of copies of data whenever it was feasible, in order to minimize data storage. However, if it came to a choice between flexibility and optimized efficiency, we invariably chose flexibility.

Our implementation of a network model has been specifically designed to run under the UNIX operating system. In keeping with the UNIX overall approach, we have designed tools which can be used to define record and set types (DDL), and to create, manipulate and delete record and set occurrences from a database (DML). For our purpose, a "tool" is a specialized function that solves a general problem and is easy to use.

Our tools are written in the C Programming Language. C is well supported under the UNIX system. The UNIX operating system itself is written in C, as is most of the UNIX applications software. C is independent of any particular machine architecture and every effort has been made to make this system
"portable". C's recursive ability has been used throughout our network system, allowing small pieces of code to perform significant tasks.

Our database system handles its own buffering for input and output. The reads and writes through the UNIX operating system are accomplished in page size blocks. Stonebraker [81] compared and contrasted the way operating systems, especially UNIX, handled buffer pool management for database systems. He concluded, "The strategy used by most DBMS's is to maintain a separate cache in user space. This buffer pool is managed by a DBMS specific algorithm." Therefore, we feel that handling our own input/output buffering is an efficiency enhancement to our system.
2. A NETWORK APPROACH DATA MODEL

We have provided a network approach, low level definition and manipulation system for a DBTG like network database, that operates under the UNIX operating system. A manipulation language has been provided to communicate with the storage and retrieval routines. However, it should be understood that the manipulation language is not written for the "user" of the database. It is assumed that another layer of software will take user commands and communicate them to this system. This software will provide a data sublanguage for the user, will parse the user commands, and then use the manipulation language to satisfy the user requests. The following is a list of the definition and manipulation primitives provided by this database management system. The primitives are explained in detail in Chapter 4.

Primitives to create record and set types:
- record type add
- set type add

Primitives to manipulate record and set occurrences:
- add record
- add member
- delete record
- delete member
- delete owner
- find record
- find owner
- find first
- find next
- change owner
- change all
- end update

2.1. MEMBERSHIP CLASS

The membership class for our system is MANUAL/MANDATORY. The MANUAL storage class implies that storing a record, R, in the database, does not
automatically insert R into a set occurrence in which it participates. To insert R into a set occurrence, an explicit "add member" command must be given. Note, however, that if R’s record type participates as an owner in any set type, S, R is automatically an owner occurrence in S. The MANDATORY removal class dictates that once an occurrence of a member, say A, has been entered into a set type, say S, as a member in a set occurrence, A can never have any existence in the database not as a member of some occurrence in S [Date 77]. Specifically, A can be moved to a new owner in S with a "change owner" or "change all" command, but may not be removed from S without being removed from every set occurrence in which it participates. Note that the removal of an owner in a set occurrence then causes the recursive removal of all its member records. This removal in the hands of the untutored could create havoc in the database. MANDATORY removal class protects the integrity of the data, in that once a record is removed all traces of it are deleted.

2.2. MODEL ENTITIES

This model uses two different entity types to represent the information contained in the database. Record types are used to represent stored user data and set types are used to store relationships between record types.

2.2.1. RECORD TYPES

Record types represent occurrences of some entity that the user expects to store and manipulate. Before a user can store a record of a specific type in the database, he must define that record type for the database management system. A record definition consists of a record type name, a field delimiter used for that record type, a specification of the number of fields the record type will contain, a
specification of the number of fields that will make up the key for the record type, and the location of the key fields in the record type. The system creates the record type by inserting the record type name with ".rf" appended to it into the binary tree containing record type definitions. The user is allowed to use variable length records within the same record type.

Once the user has defined a record type, the system will store the information about that type for future use. The information is stored in a table (implemented as a binary tree), which is used by the Database Management System. The next time the system is used, it will recall all previous record type definitions. When the user then issues an "add record" command for a particular record type, the database system fills an input/output buffer with the desired record, checks the field count on the record for an error, and builds the record's key. The key is also error checked for duplicity. The record is then passed to the input/output buffering system where it is stored at the end of a file whose name is the record type name with the suffix ".rf". As the record is passed to the buffering system, the location of the record within the file is determined. The location of a record is the block offset from the beginning of the file and then an offset to the position within the block. Once the address of the record within the file is known, the address information and the record key is stored in the record type B-tree. If the record type participates as an owner in any set type, the new record is added as an owner occurrence each such set.

In an effort to keep these records "clean", we have stored them in regular flat files in normal UNIX tradition. "Clean" means that no database pointers, id numbers, or other database management information is stored in the user record files. This allows the stored database files to be used with any other UNIX tools.
For example, a record file could be printed directly. This separation of record files from database information is one of the strengths of our system, for it is in keeping with the UNIX idea of software tools. However, the user should remember that deleted records still remain on the record files until garbage collection is done.

2.2.2. SET TYPES

Set type occurrences are represented by links stored in set type information records. The set type information allows the user to walk around the links that associate a particular owner with members in his set. Such set type information is maintained separately from record type information.

Before a user can store any set occurrences for a set type, he must define the set type. This is done with a "set add" command, followed by the set type name, the owner record type name, and the member record type name. Once the set type is defined, the database management system stores the set type information in a table along with the currency information for that set type. The set type definition must be given before any records of the set owner record type have been added to the database. The system creates a set type by inserting the set type name, with a "sf" appended to it, into the set type table (implemented as a binary tree). Once a set type has been defined in the database, it may not be removed. The next time the database is used, the database management system will recall all previous set type definitions.

After the set type has been defined, owner occurrences are automatically created whenever a record of the owner record type is added to the database. When an owner record occurrence is added to the database, a link-record for the
owner is built (See Section 4.3). This link-record is then given to the i/o buffer system to be added to a file named by the set type name with ".sf" appended to it. As the record is passed to the i/o buffer system, information is gathered on the location of the pointer record within the set type file. This location information and the owner key are stored in the set type B-tree. The link-records are stored as regular UNIX files and, therefore, can be used as arguments for other UNIX tools.

If the user wishes to add a member to a set occurrence, he must give an explicit "add member" command. The user specifies the set type and owner key of the appropriate set occurrence. Inserting the new member into a set occurrence is accomplished by building a link-record for the new member. These link-records make up a chain that connect the set occurrences, and can be traversed to obtain all the members of a set occurrence. (For exact information on link-record insertion, see Section 4.3.) The new member occurrence is added as the immediate successor to the owner. Order of retrieval is therefore LIFO (Figure 2-1).

As with the owner's link-record, the member's link-record are stored through the buffering system, on a file named by the set type name with ".sf" appended to it. Once the link-record is stored, the file offset information along with the new member key is stored in the set type B-tree.

2.3. DATA STRUCTURES EMPLOYED

We have used three major data structures: B-trees, binary trees, and doubly linked lists. Obviously, other data structures were used in the coding of the system, however, they have little impact on the design of the overall system.
### 2.3.1. B-TREE INDEXES

We used the B-tree index [Wirth 76] to store our key and offset information for both the record types and the set types. The B-tree is, de facto, the standard organization for indexes in a database system [Comer 79].

### 2.3.2. TABLE STORAGE IN BINARY TREES

We used binary trees to store the database management record type and set type tables. Separate trees were maintained for the record type table and the set type table. Each tree contains one node for each record/set type. The nodes are maintained so that at any node, the left subtree contains only record/set types whose name precedes the node's record/set type name with a string comparison; the right subtree contains only types that are greater. Each record type node contains:

- the record type name,
- the field delimiter for this record type,
- the number of fields to expect in the record,
- the number of key fields, i.e., the number of fields used to build the key,
- an array containing the position of the key fields,
- a structure of currency pointers,
pointers to the left and right record type binary subtree.

Each set type node contains:

- the set type name,
- a pointer to the record table entry for the owner record type,
- a pointer to the record table entry for the member record type,
- the file descriptor and root for the B-tree index for this set type,
- current of set type key
- a structure of currency pointers,
- pointers to the left and right set type binary subtree.

"Currency pointer" structures are used primarily by the input/output buffering routines to place records in the buffers and ultimately into files. The structures contain:

- logical file and block offsets,
- physical file and block offsets,
- final file and block offsets,
- a switch that indicates if a record has just been added to the file,
- a pointer to the file buffer for this record/set type.

The logical offsets indicate the beginning of the record that was most recently referenced for the record/set type. This supplies the current of run for this record/set type. The physical offsets indicate the end of the record that was most recently referenced for the record/set type. The final offsets indicate the end of the file for the record/set type.

When a new record or set type is defined, a node is created for it in the appropriate binary tree. At the conclusion of the database update, the pertinent information from the binary type trees is stored in a file named dbm_file. (This name is a parameter, and easily changed.) The first field on the dbm_file is the number of record types defined within the database, followed by the number of set types defined. After these come the record type information and then the set type
information. When the database management system is run, the system builds the new binary type trees from the information stored on the dbm_file. At this time the record type B-tree is opened and its file descriptor and root are stored in global storage. Again, the dbm_file is a normal UNIX file that can be examined by and used with other UNIX tools. If no record types or set types have been defined, the system will print a message informing the user of the fact.

2.3.3. EXPRESSING SET RELATIONSHIPS

Our implementation uses doubly linked lists to represent set occurrences. The front pointer of any owner or member in a set occurrence contains the key of the next owner or member in the set occurrence. The back pointer of any member contains the key of the prior member or the owner. The owner's back pointer is always null. These pointers are placed in a link-record, with the forward pointer first and backward pointer second. The link-record is then stored on the set type file, and the owner/member key and the file offset of the link-record are stored in the set type B-tree. As an example, suppose owner A1 owns members B1 and 2B. Then A1's forward pointer would be 2B and backward pointer would be null. B1's fp would be A1 and bp would be 2B. 2B's fp would be B1 and bp would be A1. In order to follow the link to the next owner or member link, the key is found in the set type B-tree, and the offset for the next link-record is obtained. When a member is deleted from a set, the previous and following member's links are reset to exclude the member, and the deleted member's key is removed from the set type B-tree. The member is also removed from the record B-tree and is then removed from any other set where the member participates as an owner or member. When an owner is deleted from the set, the owner's key is removed from the set B-tree, then removed from the record B-tree, and finally removed from any
other set where the owner participates as an owner or member. When an owner is deleted from a set, each of his members is also deleted.

2.4. INPUT/OUTPUT BUFFERING

As a time efficiency measure, we have implemented buffered input and output under the control of the database management system. Many DBMSs, including INGRES and SYSTEM R, have chosen to put a DBMS managed buffer pool in user space to reduce overhead [Stonebraker 81]. Either at the beginning of the update session for predefined types, or at the time a new record type or set type is defined, a page-size block of memory is allocated for its i/o buffering. A pointer to the buffer block is stored in the set/record type table for use by the i/o system. All reads and writes are accomplished in page-size blocks. When the i/o system is asked by the calling routines to store a record, the i/o system expects to find the record in a sending buffer. The i/o routines will then add the record to the end of the appropriate file and return the location of the record to the calling routine. This is accomplished in the buffer area for the record/set type and does not always require actual writes to the file. When the i/o system is requested by the data model to retrieve a record, the i/o system will try to fill the order from data already resident in main store in the buffer area for that record/set type. This will be possible frequently, especially when link-records are sought. When the i/o system locates the appropriate record, it is placed in a receiving buffer supplied by the calling routine. The i/o system updates the currency pointers for record/set types as it does its storage and retrieval functions.
2.5. ERROR HANDLING

All error handling is done through two routines. One error routine accepts a single string as the error message, prints the message, and returns an error indicator. The other error routine accepts an error message and a key value. Both the message and the key value is printed and an error indicator is returned. It would be very easy to change these routines so that no error message was printed, but rather a specialized code would be returned.
3. PROTOTYPE

We built a simple prototype database to be manipulated with our database management system. The prototype consists of four record types and three set types with certain relationships expressed through set occurrences.

3.1. RECORD TYPE DESCRIPTIONS AND CONTENTS

Record Type Name: faculty
Field Delimiter: *
Number of Fields: 5
Number of Key Fields: 1
Position of Key Fields: 2

Contents:

Peter*A1*10*A186*25
Bill*A2*10*2132*57
Roy*3A*10*A285*72
Jack*4A*10*1116*13

Record Type Name: student
Field Delimiter: :
Number of Fields: 4
Number of Key Fields: 1
Position of Key Fields: 3

Contents:

Mary:CAST:B1:Comp Scie
Leslie:CAST:B2:Comp Scie
John:SP:3B:PPPD
Tom:CAST:4B:Syst Soft
Mary:SP:5B:PPPD
Record Type Name: housing
Field Delimiter: *
Number of Fields: 3
Number of Key Fields: 1
Position of Key Fields: 1

Content:

405*Billings*25
216*Watson*1105

Record Type Name: courses
Field Delimiter: *
Number of Fields: 8
Number of Key Fields: 4
Position of Key Fields: 5 1 3 4

Contents:

B1*0601*81*1*875*1*D*r
B2*0601*81*2*875*1*A*nr
B1*0601*81*1*720*2*B*nr
3B*0532*81*2*875*1*A*nr
B2*0532*81*2*850*1*B*r
B1*0601*81*2*875*1*B*r
5B*0601*80*2*875*1*D*r
5B*0601*81*3*875*1*E*r
3.2. SET TYPE DESCRIPTIONS AND CONTENTS

Set Type Name: fs
Owner Record Type: faculty
Member Record Type: student
Relationship: the faculty member is the advisor of the student

Contents:

A1 owns
   B1
   3B
A2 owns
   B2
   4B
3A owns
   5B
4A has no members

Set Type Name: sc
Owner Record Type: student
Member Record Type: courses
Relationship: the student has taken the course

Contents:

B1 owns
   875B1811
   720B1812
   875B1812
B2 owns
   875B2812
   850B2812
3B owns
   8753B812
4B has no members
5B owns
   8755B802
   8755B813

Set Type Name: hs
Owner Record Type: housing
Member Record Type: student
Relationship: the student is a resident of the housing complex

Contents:

405 owns
B1
5B
216 owns
3B

See Figures 3-1 and 3-2 for a graphic representation of these relationships.
FIGURE 3-2

PROTOTYPE SET OCCURRENCES

THESIS

3-5

JANUARY 1982
4. AVAILABLE PRIMITIVES

We have made every attempt to "tool" our primitives in the UNIX tradition. We have tried to develop tools that are easy to use, work well within the overall UNIX environment, and efficiently accomplish the task for which they are designed. The manipulation language was designed to be terse, as most UNIX commands are. However, in most cases the user is allowed to be more verbose. As an example, the "ra" or record add command, is actually parsed by the leading "r". Anything else can follow within the word; thus "r", "ra", "recadd", and "recordadd" would all be accepted as a record add command.

All record type names and set type names consist of the first ten characters of the word supplied. The number ten exists as a parameter in the system and is easily changed. (UNIX allows eleven characters.) As an example, if the name RochesterInstitute was input as the name of a record type, the record type would be known to the system as RochesterI.

The maximum number of key fields for a record type in the system is set at ten, with the maximum number of characters in a key set at twenty. Again, these numbers are parameters and easily changed. If more than the maximum number of characters are entered for a key value, an error message is printed and the command is aborted. As an example, if A12345678901234567890 was entered as a key value in a command, an error message would be printed.

Some of the commands are allowed to include optional input or output file names. The file name, which can be any UNIX pathname, may be up to forty characters long; extra characters are ignored. Note: all examples in the following tool descriptions will refer to the prototype described in Chapter 3.
4.1. DATABASE DEFINITION DIRECTIVES

The database definition primitives deal with defining the record types and set types.

4.1.1. RECORD TYPE ADD

The record type add command describes a record type for the database management system. The DBMS then stores the information as a node in the binary record type tree. The call to this tool has the form:

\[ \text{ra} \ <\text{name}> \ <\text{field delimiter}> \ <\# \ of \ fields> \ <\# \ of \ key \ fields> \ <\text{position of kf}> \]

A record type add command for courses would look like:

\[ \text{ra} \ \text{courses} \ * \ 8 \ 4 \ 5 \ 1 \ 3 \ 4 \]

4.1.2. SET TYPE ADD

The set type add tool allows the user to define a set type to the database management system. The DBMS then stores the information as a node in the binary set type tree.

The call to the set type add tool has the form:

\[ \text{sa} \ <\text{name}> \ <\text{owner record type name}> \ <\text{member record type name}> \]

The set type fs would have been defined by the command:

\[ \text{sa} \ \text{fs} \ \text{faculty student} \]
4.2. DATABASE MANIPULATION PRIMITIVES

The database manipulation primitives allow the user to add items to the database, remove items from the database, find items within the database, and reorganize members within set types.

4.2.1. DATABASE EXPANSION

When an owner or a member is added to a set, a link-record is built and integrated into the appropriate set. Link records consist of back and forward pointers and are discussed in detail in Chapter 2, Section 4.3. Once the link-record is built and added to the file for the set type, the key for the owner/member along with the file offset of the link-record is stored in the set type B-tree.

4.2.1.1. ADD RECORD

The add record tool allows the user to store a record for a particular record type in the database. If that record type is the owner in any set, another tool, "add owner", is called and the new record key is automatically added as an owner in the set. The call to the add record tool has the form:

    ar <record type name> [ <input file name> ]

If the input file name is supplied, all the records on the file are added to the database. Otherwise, records are taken from the standard input until an EOF is read. An add record command for a faculty record would look like:

    ar faculty
    Peter*Al*10*A186*25
    EOF
4.2.1.2. ADD OWNER

If a newly added record is an owner in a set type, the system will automatically generate an add owner call. The add owner tool call has the form:

\texttt{ao \langle set type\rangle \langle owner key\rangle}

For the above add record to faculty, a generated add owner tool call would look like:

\texttt{ao fs Al}

4.2.1.3. ADD MEMBER

The add member tool adds a member key to a set occurrence. The system will check the member record to make sure it conforms to set and record definitions. The call to the add member tool has the form:

\texttt{am \langle member key\rangle \langle set type name\rangle \langle owner key\rangle}

A command to add student Bl to faculty A1 would be:

\texttt{am Bl fs A1}

4.2.2. DATABASE CONTRACTION

The database management system's deletion tools are a recursive delight. Since deleting a record implies possible deletion of set occurrences and vice versa, recursion is required. Each deletion tool was designed to do its own particular job and then call a friend to do the rest.
4.2.2.1. DELETE RECORD

When a record is deleted from the database, the record key and file offset are removed from the record type B-tree. Since the offsets for the actual record can no longer be referenced, the record is logically deleted from the database. However, the actual physical record will still remain on the record type file. This record then becomes the province of the garbage collection system which is discussed in detail in Chapter 5. The call to the delete record tool has the form:

    dr <record type> <key>

A delete record command to delete student Bl's record would consist of:

    dr student Bl

Once the record is logically deleted, delete record recursively searches the set type binary tree looking for sets in which this record type participates as an owner or member. If a set is found, delete owner or delete member is called to remove the discarded record key from the set. For instance, the "dr student Bl" command above would cause Bl to be removed from the fs set and the hs set as a member and the sc set as an owner. The removal from the sc set has further ramifications which are covered in detail under "delete owner".

4.2.2.2. DELETE MEMBER

The delete member tool causes a member's key and link-record offset to be removed from the set type B-tree. The pointers in the link-records of prior and post set occurrence members are reset to exclude the deleted member. The actual link-record is left on the set type file, requiring garbage collection at a later time. However, since this link-record can no longer be referenced, it is effectively
deleted from the database. The call to the delete member tool has the form:

```
dm <set type> <key>
```

A delete member command for student B1 in the fs set would consist of:

```
dm fs B1
```

First, B1’s key and link-record file offset would be deleted from the fs set type, then B1’s record would be removed from the student record file, then B1 would be removed as a member from the hs set, and finally, B1 would be removed as an owner from the sc set, triggering the deletion of three course records.

### 4.2.2.3. DELETE OWNER

The delete owner tool is probably the most powerful tool in the database management system. Somewhat like the A-bomb, the delete owner command can cause fallout to the farthest reaches of the database. When the delete owner command is given, the owner’s key and link-record file offset are removed from the set type B-tree. Next, "delete member" is repeatedly called until all of this owner’s members are deleted. Finally, "delete record" is called to delete the owner’s record from the record type file. The delete owner command has the form:

```
do <set type> <owner key>
```

The command to delete owner A1 in set fs would consist of:

```
do fs A1
```

This small, unimpressive looking command would cause A1 to be removed from the fs set. Next, B1 would be removed from the fs set and the hs set and courses
875B1811, 720B1812, and 875B1812 would be removed from the sc set and the courses record file, and B1 would be removed from the student record file. Then, 3B would be removed from the fs and hs set files, courses record 8753B812 would be removed from the sc set and the courses record file, and 3B would be removed from the student record file. Finally, A1 would be removed from the faculty record file. Obviously, when it comes to the deletion tools, looks are very deceiving, and caution is the key word.

4.2.3. DATABASE NAVIGATION

The find tools allow the user to navigate the database set occurrences. With the proper sequence of find commands, the user is able to extract information from the database. If an output file name is specified with the find command, the requested record will be output to the named file. Otherwise, the record is sent to standard output.

4.2.3.1. FIND RECORD

The find record tool will find a record given a specific record type and key. The current of run unit is set to the key of the located record. The find record command has the form:

```
fr <record type name> <key> [<output file name>]
```

The find record command:

```
fr housing 405
```

would cause the following record to be output:

```
405*Billings*25
```
4.2.3.2. **FIND OWNER**

The find owner tool finds the owner record for a member key in a particular set. The current of run unit and the current of set type is set to the owner key.

```
fo <set type> <member key> [<output file name>]
```

The find owner command for student B2 in the fs set would consist of:

```
fo fs B2
```

and would yield the record:

```
Bill*A2*10*2132*57
```

The current of run unit and the current of fs member would be: B2.

4.2.3.3. **FIND FIRST**

The find first tool finds the first member of an owner in a specific set. Both the current of run unit and the current of set type, are set to the key of the first member. The record of the first member is output. The find first command has the form:

```
ff <set type> <owner key> [<output file name>]
```

The command to find the first member of B1 in sc would consist of:

```
ff sc B1
```

and would yield the record:

```
B1*0601*81*1*875*1*d*r
```
and would leave the current of run unit and current of set type set at 875B1811.

4.2.3.4. FIND NEXT

The find next tool finds the next member in a specified set. The next member is defined to be the next member in the set occurrence of the current of set key. The find next command resets the current of run unit and the current of set to the next member's key. If the next item in the set occurrence is the owner, a message is returned. Otherwise, the record of the next member is returned. The find next command has the form:

\[ \text{fn } \langle \text{set type} \rangle \ [\langle \text{output file name} \rangle] \]

Assuming that the above "ff fs A2" command was given, followed by:

\text{fn sc}

The following record would be returned:

\text{Bl*0601*81*1*720*2*B*nr}

If the following lists of commands were given:

\text{ff fs A1}  
\text{ff sc 3B}  
\text{fn fs}  
\text{fn sc}

the output would be:

Mary:CAST:B1:Comp Scie  
3B*0532*81*2*875*1*A*nr  
John:SP:3B:PPPD  
No more members
It should be noted that with these basic find commands, the software package interacting with the user, will be able to create different find macros, and therefore, offer the user a more extensive set of find commands. Also, since the record type files are regular UNIX files, the user can simply cat or print the record type file, if he wants a list of all the records in a particular file.

4.2.4. DATABASE REORGANIZATION

The database management system will allow the user to reorganize the set occurrences within a given set. A member can be moved from one owner to another owner within a given set.

4.2.4.1. CHANGE OWNER

The change owner tool moves a member from one set occurrence to another set occurrence. This is accomplished by resetting pointers in the link-records so that the member is linked to a new owner. The change owner command has the form:

```
co <new owner key> <set type> <member key>
```

The command to change student B1 in the set fs to a new advisor 4A would be:

```
co 4A fs B1
```

After this command, A1 would have only one advisee, 3B, and 4A would now be the advisor of B1.
4.2.4.2. CHANGE ALL

The change all tool moves all the members of an owner to a new owner. This is accomplished by repeated calls to change owner until all the members are moved. The change all command has the form:

```
ca <new owner key> <set type> <old owner key>
```

The command to move all of owner 405's members in hs to owner 216 would look like:

```
ca 216 hs 405
```

After this command was executed, 216 will have B1, 3B, and 5B as members, and 405 will have no members.

4.2.5. SAY GOOD-BYE

The final tool supplied by the database management system is a tool that allows the user to gracefully exit the updating session. The exit tool has the form:

```
q
```

After the exit command, the database management system stores record and set type definitions on a file, closes all record type, set type and B-tree files, and closes down the system.
5. CONCLUSIONS AND FUTURE EXTENSIONS

With our data model, we have tried to provide a solid foundation on which a flexible, user oriented sub-schema can sit. We have attempted to tailor our database management tools to accomplish specific tasks, yet allot them the power to do that task well. Our database record files are "clean" and fit well into the overall UNIX file system. However, as with all things this side of Paradise, certain extensions and enhancements would increase the effectiveness of our database management system. Our hope is that all or part of the following extensions will be implemented at a future time.

5.1. IMPLEMENT A SUB-SCHEMA WITH A DATA SUBLANGUAGE

We need to implement a subschema definition language and a stand alone user query language. This sublanguage should offer a greatly simplified view of the database to the user.

Within the user query language, certain enhancements could be made to the database management system. "Find" commands defined at this level could take some of the navigational tediousness away from the user. For example, the query language might allow the user to just say "find next" without presetting the current of set with a "find first". By keeping track of previous commands, the software could change the "find next" to a "find first". Also, a find macro might be nice. When a user wants to walk around a set occurrence, instead of repeated calls to "find next", the macro could expand a command, say "find all", to a "find first" followed by as many calls to "find next" as are needed. Perhaps, the software might even provide some safety alarms for the delete commands, to save the user from himself. Although it is not immediately apparent how this could be done, it
is a matter that should be pursued. Otherwise, the insatiable appetite of the delete commands will be a constant threat to the unwary user.

The sub-schema would have to implement some sort of field manipulation for the user. Our database management system deals in records. An add command, adds a whole record. A find command, finds a whole record. There will be times when the user wants to see a specific field of a record or will want to change a field. Obviously, we can't allow the user to change key fields, but he should be allowed to change non key fields. This option should be supplied to the user by the sub-schema.

5.2. TAKING OUT THE TRASH

When a delete of a set owner/member or the delete of a record is enacted under the database management system, the set occurrence or record is logically deleted from the database. That is, any reference to the deleted item will not be satisfied. However, the actual physical file record is not deleted. Over time, the database management system will cease to operate efficiently. Storage will be wasted on inaccessible and useless data. Time will be wasted since the buffering system will be required to do extra reads and writes to carry the defunct information. Obviously, we will want to create some vehicle for reclaiming this storage. There are many ways this garbage collection could be implemented.

The most elegant way to do garbage collection within the database management system would be to garbage collect while the user was inputing commands or perusing recently returned records. Some type of timing element would have to be built, so that every so often the garbage collection routine would check to see if the user had input a new command. This garbage collector would clean a certain
section of a file by moving the good records as far as possible toward the front of the file. The B-tree offset would then have to be reset. The real key to this approach would be the timing. If timed improperly, the garbage collector could clean to the end of file, be interrupted by the user who adds a new record to the end of the file, then start up again and blow away the recently added record. Granted, this approach would be a challenging design problem. However, it would buy you free garbage collection during user "think" time.

Another approach to the garbage collection problem is a B-tree walk and move procedure. A tool could be built that would visit each node in the B-tree index for a file. With the file offset gathered from the B-tree, the record could be moved to a new file and the offsets readjusted to fit the record's new placement. Once the new "clean" file is built, the old file space is returned to the system. This tool would have to be run without interruption and could tend to be expensive time wise if the database files got very large. However, it has the advantage of providing a means to the user to be absolutely certain that only valid records appear on a record file at a given time. Once the user ran this tool, he could "cat" the record file and have a copy of all the records of a certain record type.

Yet another approach to the refuse problem for record files would be the inspect and slide function. Armed with the record type definition, the garbage collection machine could move down the record file, inspecting each record as it moved. The key for the record would be recreated using the record type definition. Then the B-tree would be searched for that key. If the key was present in the B-tree, the record would slide as far forward on the file as possible, and then the offsets in the B-tree would be reset to reflect the new location. If the key
was not in the B-tree, the record would be overlaid when a valid record from farther down the file was slid over it. This method works well for the record type files; however, it will not work for the set type files, since the link-records carry no indication of their owners. At the same time, the link-records will generally take up less space than the record type records, and it is doubtful that a user would ever want to "cat" the link-records. Therefore, garbage collecting just the record files may be enough in some instances.

There are undoubtedly many more ways to collect wasted file space. The designer of the garbage collection tool will have to weigh the merits of each possibility, and implement the most reasonable approach. However, we know that garbage collection can be effectively done on the database files.

5.3. DELETION OF RECORD AND SET TYPES

It would be nice if the database management system allowed the user to delete a record or set type once it had been defined. Deletion of a record or set type would automatically carry with it deletion of any existing records or sets of that type. This deletion could be accomplished by just removing the name of the record/set type from the binary tree node where it is stored. The record/set type would then be logically deleted. With minor changes, the end of update routine could be designed to ignore any record or set type with a null name and not store it on the dbm_file. The next time the database management system was started, the deleted record or set type would not be in the binary tree. This approach gets complicated if the user is allowed to delete a record or set type anytime, because then the database management system has to contend with searching binary trees with null names scattered about. However, if the the user is only allowed to delete a record or set type at the end of an updating session, the implementation
becomes trivial.

5.4. SPIFFY OPTIONS

There are several options that would be convenient for the user and relatively easy to implement. It might be nice for the user to be able to get a copy of all deleted records. This could be accomplished by using a find to get the record for the user and then doing the delete. It could be implemented as a macro at either the schema or sub-schema level. A tool to allow the user to question the DBMS about a record type or set type definition would be useful. Perhaps the user would like to know what the field delimiter is for a particular record type. Along the same line, the user might want to change the field delimiter for a particular record type. In terms of implementation, these options are time consuming rather than difficult.

A very desirable tool would be a listing tool. The tool would allow the user to walk around a set type and list all the owners with their members. A variation of this tool, could list the record type files, with the fields listed in columns and the key fields indicated. A listing of all the set type and/or record type definitions would be useful.

Undoubtedly, pages and pages could be written on possible extensions the would be helpful to the user. Many users will have special requirements that make some options more important than others. However, as in all things, it comes down to time and resources. We have made every effort to make our data model flexible enough to handle future options and extensible enough to make the implementation easy. It is our fondest hope that this embryo will grow and thrive, and that it will eventually mature into a contributing adult in the field of database
management systems.
BIBLIOGRAPHY


Joy, William, "An Introduction To The C Shell"


Kernighan and Ritchie, "UNIX Programming - Second Edition"

Lehman and Yao, "Efficient Locking For Concurrent Operations On B-trees", 1979


BIBLIOGRAPHY


Wirth, Niklaus, Algorithms + Data Structures = Programs Prentice-Hall, Inc., 1976
APPENDIX A: LISTING OF DATABASE MANAGEMENT PROGRAMS

FILES LISTED

main.c
definitions
p_c_dec
parse_dbup
rec_t
set_t
init_concl
utility
io
dbup_c_dec
dbup
/* This file contains the main driver for the database management system. */

#include <stdio.h>
#include "definitions"
#include "parse_dbup"
#include "rec_t"
#include "set_t"
#include "init_concl"
#include "utility"
#include "lo"
#include "dbup"

main()
{
    if (init() == OK)
    {
        dbupdate();
        store_dbmfs();
    }
}

```c
struct ci /*currency information for record or set types*/
{
    int fd;
    long log_fo;
    long phy_fo;
    long final_fo;
    int log_bo;
    int phy_bo;
    int final_bo;
    int add_on;
    char file_buf[BLOCKSIZE];
};

struct rec_type /*structure for information for a record type*/
{
    char name[MAX_TNAME + 1];
    char fdlim; /*field delimiter for this record type*/
    int nosfield;
    int noskey;
    int keyf[MAX_KEY_FIELDS];
    struct ci rci; /*record currency information*/
    struct rec_type *left;
    struct rec_type *right;
};

struct set_type /*current of set type*/
{
    char sname[MAX_TNAME + 1];
    struct rec_type *owner;
    struct rec_type *member;
    char cos [MAX_KEY + 1];
    int stfd;
    long sbtroot;
    struct ci sci; /*set currency information*/
    struct set_type *sleft;
    struct set_type *sright;
};

struct ptrs
{
definitions 101 1 long btp; /*struct used for communication with btree routines*/
definitions 102 1 int bts;
definitions 103 1 long btd;
definitions 104 1 int btb;
definitions 105 1 }

definitions 106 0 /*Global Declarations*/
definitions 107 0 struct rec_type *rt_root = NULL; /*root for record type*/
definitions 108 0 struct rec_type *ct_root = NULL; /*current of run record type*/
definitions 109 0 struct set_type *st_root = NULL; /*root for set type*/
definitions 110 0 struct set_type *co_root = NULL; /*current of run set type*/
definitions 111 0 char cor_unit[MA\_KEY+1]; /*current of run unit key*/
definitions 112 0 int cnt_root=0; /*number of record types in db*/
definitions 113 0 int cnt_st=0; /*number of set types in db*/
definitions 114 0 long rbtree; /*root for record btree*/
definitions 115 0 int rbtdf; /*file descriptor for record btree*/
/*This file contains common declarations for the parse routines.*/

char key[MAX_KEY + 1];
char sname[MAX_TNAME + 1];
char rname[MAX_TNAME + 1];
char fname[MAX_TNAME + 1];
extern struct set_type *cor_st;
extern struct rec_type *cor_rt;
extern char cor_unit[MAX_KEY + 1];
struct rec_type *prname(), *pr;
struct set_type *psname(), *ps;
int spec;
/Dbupdate parses user commands and calls appropriate subroutines*/

#define DEBUG

int more=TRUE;
char cmd[MAX_CMD + 21];
int i,cond,status;
char c;

if (DEBUG) 
 printf("d dbupdate\n");
#endif

while (more)
{
    printf("%c", PROMPT);
    if (scanf(FMT3, cmd) != NULL)
        switch(*cmd)
        {
            case 'q':  /*end of update*/
                printf("end of update\n");
                more = FALSE;
                status = LP;
                break;
            case 'r':  /*record type add*/
                status = rect_add();
                break;
            case 's':  /*set type add*/
                status = psett_add();
                break;
            case 'a':
                switch (*(cmd+1))
                {
                    case 'o':  /*owner add*/
                        status = pownadd();
                        break;
                    case 'm':  /*member add*/
                        status = pmebadd();
                        break;
                    case 'r':  /*record add*/
                        status = precadd();
                        break;
                    default:
                        status = ERROR;
                        break;
                    }
                break;
            }
case 'c':
    status = pchange(*cmd+1); /*parse change command*/
    break;
  case 'd':
    switch (*cmd+1)
    {
      case 'm': /*delete member*/
        status = pdel(*cmd+1);
        break;
      case 'o': /*delete owner*/
        status = pdel(*cmd+1);
        break;
      case 'r': /*delete record*/
        status = 'precdele();
        break;
      default:
        cond = ERROR;
        break;
    }
    break;
  case 'f':
    switch (*cmd+1)
    {
      case 'r': /*find record*/
        status = precfind();
        break;
      default:
        status = pfnd(*cmd+1);
        break;
    }
    break;
  default:
    cond = ERROR;
    break;
} else
    cond = ERROR;
if (cond == ERROR) {
  printf("illegal command\n");
  cond = OK;
  status = LNP; /*full line not parsed*/
}
if (status != LP)
    while ((c = getc(stdin)) != END_OF_LINE)
        ; /*scan the rest of the line*/
    }
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse_dbup</td>
<td>101</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>parse_dbup</td>
<td>102</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>parse_dbup</td>
<td>103</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>parse_dbup</td>
<td>104</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
parse_dbup Mon Sep 27 14:43:37 1982 Page 4

fil* line level source

parse_dbup 106 0 /*Prect_add parses the command line for a record type add command, gathering
parse_dbup 107 0 information. If the information is complete, rect_add is called.*/
parse_dbup 108 0
parse_dbup 109 0
parse_dbup 110 0
parse_dbup 111 0
parse_dbup 112 1
parse_dbup 113 1
parse_dbup 114 1
parse_dbup 115 1
parse_dbup 116 1
parse_dbup 117 1
parse_dbup 118 1
parse_dbup 119 1
parse_dbup 120 1
parse_dbup 121 1
parse_dbup 122 1
parse_dbup 123 1
parse_dbup 124 1
parse_dbup 125 1
parse_dbup 126 1
parse_dbup 127 1
parse_dbup 128 1
parse_dbup 129 1
parse_dbup 130 1
parse_dbup 131 2
parse_dbup 132 2
parse_dbup 133 2
parse_dbup 134 2
parse_dbup 135 2
parse_dbup 136 2
parse_dbup 137 2
parse_dbup 138 3
parse_dbup 139 3
parse_dbup 140 3
parse_dbup 141 3
parse_dbup 142 4
parse_dbup 143 4
parse_dbup 144 4
parse_dbup 145 4
parse_dbup 146 3
parse_dbup 147 3
parse_dbup 148 2
parse_dbup 149 1
parse_dbup 150 0
parse_dbup 151 0
parse_dbup 152 0
parse_dbup 153 0
parse_dbup 154 0

ifdef DEBUG
parse_dbup 122 1 printf("e prect_add\n");
endif

parse_dbup 125 1
parse_dbup 126 1
if (prname(rname, &spec) == NULL &amp; spec != PRESENT)
parse_dbup 127 1 return(error( "usage: a new-rec-type-name #fields #key-fields list-key-fields")
parse_dbup 128 1 else if (jumpp() == PRESENT || scanf(" %c , &fdlm) == NULL)
parse_dbup 129 1 return(error("missing field delimiter");
parse_dbup 130 1 else
parse_dbup 131 2 /*read and validate key field information*/
parse_dbup 132 2 scanf(" %d %d", nosfld, noskeyf);
parse_dbup 133 2 if (noskeyf > MAX_KEY_FIELDS || noskeyf < 1 || noskeyf > nosfld)
parse_dbup 134 2 return(error("illegal number of key fields");
parse_dbup 135 2 else if (nosfld < 1)
parse_dbup 136 2 return(error("illegal number of fields");
parse_dbup 137 2 else
parse_dbup 138 3 for (i=0;i&lt;MAX_KEY_FIELD; i++)
parse_dbup 139 3 *(keyf+i) = 0; /*zero key fld array*/
parse_dbup 140 3 for (i=0; i &lt; noskeyf; i++)
parse_dbup 141 3 {
parse_dbup 142 4 scanf(" %d", keyf+i);
parse_dbup 143 4 if (*(keyf+i) &lt; 0 || *(keyf+i) &gt; nosfld)
parse_dbup 144 4 return(error("illegal key field specification");)
parse_dbup 145 4 }
parse_dbup 146 3 return(rect_add(rname, fdlm, nosfld, noskeyf, keyfi)); /*add the record ty
/*Pset_add parses the command line for a set type add command, gathering
   information. If the information is complete, set_add is called.*/

pset_add()
{
    char stname[MAX_TNAME + 1];
    char owner[MAX_TNAME + 1];
    struct set_type *psname();
    struct rec_type *prname();
    int spec;

#ifdef DEBUG
    printf("e pset_add\n");
#endif

if(psname(stname,$spec) == NULL & & spec != PRESENT)
    return(error("usage: sa new-set-name owner-name member-name"));
else
{
    /*scan owner and member names*/
    if (prname(owner,$spec) == NULL & & spec != PRESENT !)
        prname(member,$spec) == NULL & & spec != PRESENT)
        return(error("missing owner or member name");
    else
        return(set_add(stname,owner,member)); /*add new set type*/

return(succ

parse_dbup Mon Sep 27 14:43:37 1892 Page 5

file line level source

parse_dbup 156 0
parse_dbup 157 0
parse_dbup 159 0
parse_dbup 155 0
parse_dbup 160 0
parse_dbup 161 0
parse_dbup 162 1
parse_dbup 163 1
parse_dbup 164 1
parse_dbup 165 1
parse_dbup 166 1
parse_dbup 167 1
parse_dbup 168 1
parse_dbup 169 1
parse_dbup 170 1
parse_dbup 171 1
parse_dbup 172 1
parse_dbup 173 1
parse_dbup 174 1
parse_dbup 175 1
parse_dbup 176 1
parse_dbup 177 2
parse_dbup 178 2
parse_dbup 179 2
parse_dbup 180 2
parse_dbup 181 2
parse_dbup 182 2
parse_dbup 183 2
parse_dbup 184 1
parse_dbup 185 0
parse_dbup 186 0
parse_dbup 187 0
parse_dbup 188 0
parse_dbup 189 0
`file` line level source

```
parse_dbup   191  0 */Pownadd parses the command line for an owner add command, sets the current
parse_dbup   192  0 of run pointers, and calls ownadd*/
parse_dbup   193  0
parse_dbup   194  0
parse_dbup   195  0 pownadd()
parse_dbup   196  0 {
parse_dbup   197  1 #include "p_c_dec" /*include common parse declarations*/
parse_dbup   198  1
parse_dbup   199  1 #ifdef DEBUG
parse_dbup   200  1 printf("e pownadd\n");
parse_dbup   201  1#endif
parse_dbup   202  1
parse_dbup   203  1
parse_dbup   204  1 if ((ps = psname(sname,Spec)) == NULL &&
parse_dbup   205  1 spec != PRESENT) /*scan the set type name*/
parse_dbup   206  1 return(error("missing set name"));
parse_dbup   207  1 else if (ps == NULL)
parse_dbup   208  1 return(error("set type not defined"));
parse_dbup   209  1 else if (pkey(key) == ERROR) /*scan occurrence key*/
parse_dbup   210  1 return(0);
parse_dbup   211  1 else
parse_dbup   212  1 {
parse_dbup   213  2 cor_st = ps; /*set the cor pointers*/
parse_dbup   214  2 strcpy(cor_unit,key);
parse_dbup   215  2 return(ownadd()); /*add an owner set occurrence*/
parse_dbup   216  2 }
parse_dbup   217  1 }
parse_dbup   218  0
parse_dbup   219  0
parse_dbup   220  0
parse_dbup   221  0
parse_dbup   222  0
```
parse_dbup    Mon Sep 27 14:43:37 1982       Page 7

file        line       level      source
parse_dbup   224      0          */Pmebadd parses the command line for an add member occurrence command, sets
                the current of run pointers, and calls member add.*/
parse_dbup   225      0
parse_dbup   226      0
parse_dbup   227      0
parse_dbup   228      0       pmebadd()
parse_dbup   229      0       {
parse_dbup   230      1       #include "p_c_dec"
                char mkey[MAX_KEY + 1];       /*common parse declarations*/
parse_dbup   231      1
parse_dbup   232      1
parse_dbup   233      1       ifndef DEBUG
parse_dbup   234      1       printf(" e pmebadd\n");       .
parse_dbup   235      1       endif
parse_dbup   236      1
parse_dbup   237      1
parse_dbup   238      1       if (pkey(mkey) == ERROR)       /*parse member key*/
parse_dbup   239      1       return(ERROR);
parse_dbup   240      1       /*parse record type name*/
parse_dbup   241      1       else if ((ps = psname(sname,&spec)) == NULL & spec != PRESENT)
parse_dbup   242      1       return(error("missing record type"));
parse_dbup   243      1       else if (ps == NULL)
parse_dbup   244      1       return(error("set type not defined"));
parse_dbup   245      1       else if (pkey(key) == ERROR)       /*parse owner key*/
parse_dbup   246      1       return(ERROR);
parse_dbup   247      1       else
parse_dbup   248      1       {
parse_dbup   249      2       cor_st = ps;       /*set currency pointers*/
pitch_dbup   250      2       strcpy(cor_unit,key);
pitch_dbup   251      2       return(mebadd(mkey));
parse_dbup   252      2
parse_dbup   253      1
}
parse_dbup   254      0
parse_dbup   255      0
parse_dbup   256      0
parse_dbup   257      0
parse_dbup   258      0
parse_dbup  260  0  /*Precadd parses the command line for a record occurrence add, sets the
  currency pointers, and calls recadd.*/
parse_dbup  261  0  precadd()
parse_dbup  262  0
parse_dbup  263  0
parse_dbup  264  0
parse_dbup  265  0
parse_dbup  266  1  FILE *ifp;
parse_dbup  267  1  #include p_c_dec  /*common parse definitions*/
parse_dbup  268  1  char c;
parse_dbup  269  1
parse_dbup  270  1
parse_dbup  271  1  #ifdef DEBUG
parse_dbup  272  1  printf("e precadd\n");
parse_dbup  273  1  #endif
parse_dbup  274  1
parse_dbup  275  1
parse_dbup  276  1  if ((pr = prname(rname,&spec)) == NULL & & spec != PRESENT)
parse_dbup  277  1    return(error("missing record type"));
parse_dbup  278  1  else if(pr == NULL)
parse_dbup  279  1    return(error("record type not defined"));
parse_dbup  280  1  else if(pfname (fname, READ, &ifp) == ERROR) /*parse file name*/
parse_dbup  281  1  return(ERROR);
parse_dbup  282  1  else
parse_dbup  283  1    {    cor rt = pr;    /*set currency pointers*/
parse_dbup  284  2    while ((c = getc(stdin)) != END_OF_LINE)
parse_dbup  285  2    ;    /*scan rest of line*/
parse_dbup  286  2    recadd(ifp);    /*add record occurrence*/
parse_dbup  287  2    return(LP);    /*have read past end of line*/
parse_dbup  288  2
parse_dbup  289  2
parse_dbup  290  1  }
parse_dbup  291  0
parse_dbup  292  0
parse_dbup  293  0
parse_dbup  294  0
parse_dbup  295  0
parse_dbup  Mon Sep 27 14:43:37 1982  Page 9

file

line level source

parse_dbup  297  0  /* Precdel parses the command line for a record occurrence delete, sets the
currency pointers, and calls record occurrence delete.*/
parse_dbup  298  0
parse_dbup  299  0
parse_dbup  300  0
parse_dbup  301  0  precdel()
parse_dbup  302  0  {
parse_dbup  303  1  #include "p_c_dec"
parse_dbup  304  1
parse_dbup  305  1  #ifdef DEBUG
parse_dbup  306  1  printf("e precdel\n");
parse_dbup  307  1  #endif
parse_dbup  308  1
parse_dbup  309  1
parse_dbup  310  1
parse_dbup  311  1  /*parse record type name*/
parse_dbup  312  1
parse_dbup  313  1  if ((pr = prname(rnameA, &spec)) == NULL & spec != PRESENT)
parse_dbup  314  1    return(error("missing record type"));
parse_dbup  315  1  else if (pr == NULL)
parse_dbup  316  1    return(error("record type not defined"));
parse_dbup  317  1  else if (pkey(key) == ERROR)
parse_dbup  318  1    return(ERROR);
parse_dbup  319  1  }
parse_dbup  320  2
parse_dbup  321  2  strcpy(cor_unit, key);  /*set currency pointers*/
parse_dbup  322  2
parse_dbup  323  1  }
parse_dbup  324  0
parse_dbup  325  0
parse_dbup  326  0
parse_dbup  327  0
parse_dbup  328  0
/ *Pdel parses a command line for an owner or member occurrence delete, sets
the currency pointers, and calls owner delete or member delete depending
on the cmd parameter.* /

pdel(cmd)
char cmd;

#include "p_c_dec" /*common parse declarations*/

 ifdef DEBUG
 printf("e pdel\n");
 endif

 if ((ps = psname(sname,$spec)) == NULL & & spec != PRESENT)
   return(error("missing record type"));
 else if (ps == NULL)
   return(error("set name undefined"));
 else if (pkey(key) == ERROR)
   return(ERROR);
 else
   { cor_st = ps;    /*set currency pointers*/
     strcpy (cor_unit,key);
     if (cmd == "o")    /*call owner occurrence delete*/
       return (owndel());
     else
       return (mebdel(NEWL,NEW));
   }

return (0);


/*Preclnd parses the command line for a find record occurrence command, sets
  currency pointers, and call record find*/

precfind()
{
  #include "precde.c"
  FILE *ofp;
  
  #ifndef DEBUG
  printf ("`e precfind\n");
  #endif

  if ((pr = prname(rname,&spec)) /*parse record type name*/
      return(error("missing record type"));
  else if (pr == NULL)
    return(error("record type not defined"));
  else if (pkey(key) == ERROR || prname(fname,"APPEND",&ofp) == ERROR)
    return(ERROR);
  else
    {
      cor.rt = pr; /*set currency pointers*/
      strcpy (cor_unit,key);
      return (recfind(ofp));
    }
  }
/* find parses a command line for a find owner, a find first member, or a find next command, sets currency pointers, and appropriate find routines.*/

pfnd (cmd)
char cmd;

#include "p_c_dec"

FILE *ofp; /*output file descriptor*/

#endif DEBUG
printf ("e pfnd\n");

/* parse set name, key, and optional output file name*/

if ((ps = psname(sname, &spec)) == NULL || spec != PRESENT)
    return (error("missing set type"));
else if (ps == NULL)
    return (error("set type undefined"));
else if (cmd == 'n')
    if (pfnd (fname, APPEND, &ofp) == ERROR)
        return (ERROR);
    else
        return (nextfind (ofp));

else if (pkey(key) == ERROR || pfnd (fname, APPEND, &ofp) == ERROR)
    return (ERROR);

else
    cor_st = ps; /*set currency pointers*/
    strcpy (cor_unit, key);
    switch (cmd)
    {
    case 'f': /*find first member*/
        return (mebfnd (ofp));
    case 'o':
        return (ownfind (ofp));
    default:
        return (error("illegal command"));
    }

/* handle find next command*/
parse_dbuf 453 0 /*Parse parses the command line for a change all or change owner command, 
sets the currency pointers, and calls the appropriate change routine.*/
parse_dbuf 454 0
parse_dbuf 455 0
parse_dbuf 456 0 
parse_dbuf pchange (cmd)
parse_dbuf 457 0 char cmd;
parse_dbuf 458 0 {
#include "p_c_dec"
parse_dbuf 459 1 char newokey[MAX_KEY + 1]; /*new owner/member key*/
parse_dbuf 460 1
parse_dbuf 461 1
parse_dbuf 462 1 #ifdef DBUG
parse_dbuf 463 1 printf ("e pchange\n");
parse_dbuf 464 1 #endif
parse_dbuf 465 1
parse_dbuf 466 1
parse_dbuf 467 1 if (pkey(newokey) == ERROR) /*parse new owner/member key*/
parse_dbuf 468 1 return(ERROR);
parse_dbuf 469 1 /*parse set type name*/
parse_dbuf 470 1 else if ((ps = psetname(sname,&spec)) == NULL && spec != PRESENT)
parse_dbuf 471 1 return(error("missing set type name"));
parse_dbuf 472 1 else if (ps == NULL)
parse_dbuf 473 1 return(error("set type undefined");)
parse_dbuf 474 1 else if (pkey(key) == ERROR) /*parse old owner/member key*/
parse_dbuf 475 1 return(ERROR);
parse_dbuf 476 1
parse_dbuf 477 1
parse_dbuf 478 1
parse_dbuf 479 1 else
parse_dbuf 480 1 {
parse_dbuf 481 2 cor_st = ps; /*set currency pointers*/
parse_dbuf 482 2 strcpy(cor_unit,key); /*set at old owner/member*/
parse_dbuf 483 2 switch(cmd)
parse_dbuf 484 3 { /*call appropriate change routine*/
parse_dbuf 485 4 case 'a': /*change all members to new owner*/
parse_dbuf 486 4 return(cngall(newokey));
parse_dbuf 487 4 case 'o': /*change member to new owner*/
parse_dbuf 488 4 return(cgown(newokey));
parse_dbuf 489 4 default:
parse_dbuf 490 4 return(error("illegal command");)
parse_dbuf 491 3 }
parse_dbuf 492 2 }
parse_dbuf 493 1
parse_dbuf 494 0
parse_dbuf 495 0
parse_dbuf 496 0
parse_dbuf 497 0
parse_dbuf 498 0
parse_dbuf 499 0

/*parse utility routines*/
/*Prname scans standard input for a string, truncates the string to MAX_NAME
   and places it in string passed in as a parameter. Prname searches the
   record type tree for the record name indicated by the string.
   prname returns: NULL if the string is empty
   NULL if the string is not a name in the record type tree
   pointer to record type if name is in the record type tree*/

define **debug**

```c
struct rec_type *prname(rname, spec)
  char *rname;
  int *spec;
  {
    extern struct rec_type *rt_root;
    struct rec_type *rt_search();
  }
define **debug**  
.printf ("e prname \n");
  return(rt_search(rt_root, rname));
  return(NULL);
```
/*psname scans standard input for a string, truncates the string to MAX_TNAME, 
and places the string into the parameter passed into the routine. Psname 
searches the set type tree for the set name indicated by the string. 
psname returns: NULL if the string is empty 
NULL if the string is not a name in the set tree pointer to set type if name is in set tree*/

struct set_type *psname (stname,spec)
    
    char *stname;
    int *spec;

    {
        extern struct set_type *st_root;
        struct set_type *st_search();

        #ifdef DEBUG
        printf("entering set type parse\n");
        #endif

        if (**spec = jmpup()) == PRESENT)
            
            scanf(FMT1, stname); /*scan name*/
            return(st_search(st_root, stname));
        
        else
            return(NULL);
    }
parse_dbup  Mon Sep 27 14:43:37 1992  Page 16

FILE  line  level  source

parse_dbup  567  0  /* fname scans a file name. If the file name is null, fp is set to stdin
            or stdout depending on the mode. Otherwise, the file is opened. */
parse_dbup  568  0
parse_dbup  569  0
parse_dbup  570  0  char *mode;
parse_dbup  571  0
parse_dbup  572  0  char *fname;
parse_dbup  573  0  FILE **fp;
parse_dbup  574  0  {
            FILE *fopen();
parse_dbup  575  1
parse_dbup  576  1  #ifdef DEBUG
parse_dbup  577  1  printf("entering parse file name\n");
parse_dbup  578  1  #endif
parse_dbup  579  1
parse_dbup  580  1
parse_dbup  581  1  if (jumpup() != PRESENT)
parse_dbup  582  1     if (strcmp (mode, READ) == MATCH)
parse_dbup  583  1         *fp = stdin;
parse_dbup  584  1     else *fp = stdout;
parse_dbup  585  1  else
parse_dbup  586  1     {
parse_dbup  587  2
parse_dbup  588  2     scanf (FMT1, fname);
parse_dbup  589  2     if ((*fp = fopen (fname, mode)) == NULL)
parse_dbup  590  2         return (error("unable to open file"));
parse_dbup  591  2
parse_dbup  592  1     return (OK);
parse_dbup  593  1
parse_dbup  594  0
parse_dbup  595  0
parse_dbup  596  0
parse_dbup  597  0
parse_dbup  598  0
parse_dbup  599  0
file

date:    Mon Sep 27 14:43:37 1982
author:  Page 17

parse_dibup

/*pkey parses an input key. If key entered is too long an error message is
 returned. Otherwise, OK is returned.*/

pkey (key)
char *key;
{
    char inkey[MAX_KEY + 6];
    #ifdef DEBUG
        printf ("entering pkey\n");
    #endif
    if (jumpup() == PRESENT)
        { 
            scanf(FMT2, inkey);
            if (strlen(inkey) > MAX_KEY)
                return(error("input key too long"));
            else
                { 
                       strcpy(key, inkey);
                    return (OK);
                }
        }
    #else
        return(error("missing key"));
    }

parse_dibup

file
/*Jumpup scans blanks and tabs, and returns Present(1) if the next character is not a new line and NOTPRESENT if the next character is a new line.*/
jumpup()
{
    char c;
    #ifdef DEBUG
        printf("e_jumpup\n");
    #endif
    while (((c = getc(stdin)) == BLANK) || (c == TAB))
    {
        /*skip blanks and tabs*/
        ungetc(c, stdin);
    }
    if (c == '`
')
        return (NOTPRESENT);
    else
        return (PRESENT);
}
file

/*This file contains the routines to build the record type tree, insert a
record type into the tree, and search the tree for a particular record type.*/

/*MAD 10-81*/

/*Brec_tree reads the data base management file and builds a tree contain-
ing the information on the different record types in the data base*/

brec_tree (fp,number)

FILE *fp;

int number;

int i;

#endif DEBUG

printf("entering brec_tree\n");

#endif
rec_t 29 0 /*Rec_tt will scan the dbm file and build a record type and place it in
rec_t 30 0 the record type b-tree. Returns -1 if error occurs.*/
rec_t 31 0
rec_t 32 0 rec_tt (p)
rec_t 33 0 FILE*fp; /*dbm file pointer*/
rec_t 34 0 {
    int i,cond;
    extern struct rec_type *rt_root;
    struct rec_type r;
    struct rec_type *rt_insert();
    rec_t 39 1 /*read information for one record type*/
    rec_t 40 1 fscanf(fp,"%s %c %d %d", r.name, &r.fdlim, &r.nosfield, &r.noskey);
    rec_t 41 1 for (i=0; i<r.noskeyf; i++)
        fscanf(fp,"%d ", r.keyf+l);
    rec_t 42 1 if ((r.rc1.fd = openfile(r.name, RT)) == ERROR)
        return(RETURN);
    rec_t 43 1 else
        fscanf(fp,"%ld %d\n", &r.rc1.final_fo,&r.rc1.final_bo);
    rec_t 44 1 /*(add node to tree*/
    rec_t 45 2 rt_root = rt_insert(rt_root, &r, &cond);
    rec_t 46 2 if (cond == MATCH)
        return(error("unable to build record type tree");)
    rec_t 47 2 else
        return(OK);
    rec_t 48 2 }
    rec_t 49 1
    rec_t 50 0
    rec_t 51 0
    rec_t 52 0
    rec_t 53 0
    rec_t 54 0
    rec_t 55 0
    rec_t 56 0
    rec_t 57 0
    rec_t 58 0
    rec_t 59 0
    rec_t 60 0
    rec_t 61 0
/*rt_insert inserts a record type into the record type b-tree. If already present, condition is set to error. A pointer to the root is returned if record type is inserted.*/

struct rec_type *rt_insert(prec, r, cond)
struct rec_type *prec;
struct rec_type *r;
int *cond;
{
    ifdef DEBUG
    printf("e rt_insert\n");
    endif

    if(prec == NULL)
    {
        prec = RTALLOC;
        strcpy(prec->name, r->name);
        prec->fdlim = r->fdlim;
        prec->nosfield = r->nosfield;
        prec->noskeyf = r->noskeyf;
        arrcpy(prec->keyf, r->keyf, MAX_KEY_FIELDS);
        prec->rci.fd = r->rci.fd;
        prec->rci.phy_fd = 0L;
        prec->rci.phy_fd = 0;
        prec->rci.log_bo = 0;
        prec->rci.final_bo = 0;
        prec->rci.final_bo = 0;
        prec->rci.add_on = FALSE;
        prec->left = prec->right = NULL;
        *cond = NO_MATCH;
    }
    else if (!(*cond = strcmp(r->name, prec->name)) == MATCH)
        error("record type already defined");
    else if (*cond < 0)
        /*go to left subtree*/
        prec->left = rt_insert(prec->left, r, cond);
    else
        /*go to right subtree*/
        prec->right = rt_insert(prec->right, r, cond);
    return(prec);
rec_t 109 0 /*Rect_add will add a record type to the data base*/
rec_t 110 0
rec_t 111 0 rec_t_add (rname, rfdlim, nosfld, noskey, keyf)
rec_t 112 0 char *rname;
rec_t 113 0 char rfdlim;
rec_t 114 0 int nosfld, noskey;
rec_t 115 0 int *keyf;
rec_t 116 0 {
    struct rec_type r;
    int 1, cond;
    extern struct rec_type *rt_root;
    extern cnt rt;
    extern long rbroot;
    extern int rbfd;
    long initb();
    int verbose = NONE;
    struct rec_type *rt_insert();
    struct rec_type *rt_search();

    #ifdef DEBUG
    printf("e rect_add\n");
    #endif
    printf("e rect_add\n");
    if (cnt rt == NONE)
    {
        /*create btree for record keys and offsets*/
        if ((rbfd = creat(RBT, PMODE)) == ERROR)
            return(error("unable to create dbm.rec.bt file"));
        buildb(rbfd, SIZE, verbose);
        close(rbfd);
        if ((rbfd = open(RBT, RW)) == ERROR)
            return(error("unable to open dbm.rec.bt file"));
        rbroot = initb(rbfd);
    }
    if (rt_search(rt_root, rname) != NULL)
        return(error("record type already defined"));
    if ((r.rci.fd = create_file(rname, RT)) == ERROR)
        return(ERROR);
    if (else)
    {
        strcpy(r.name, rname);
        r.fldim = rfdlim;
        r.nosfield = nosfld;
        r.noskey = noskey;
        strcpy(r.keyf, keyf, MAX_KEY_FIELDS);
        r.rci.final fo = 0;
        r.rci.final bo = 0;
        rt_root = rt_insert(rt_root, &r, &cond);
        if (cond == MATCH)
            return(ERROR);
        else
file   line   level   source
rec_t  159   2
rec_t  161   3
rec_t  161   3
rec_t  162   3
rec_t  163   2
rec_t  164   1
rec_t  165   0
rec_t  166   0
rec_t  167   0
rec_t  168   0
rec_t  169   0

cnt_rt += 1;
return(OK);
/*rt_search searches the record_type b-tree for a specific occurrence of a
record type name. If present, a pointer to the set type is returned.
Otherwise, null is returned*/

struct rec_type *rt_search(prec, rname)
struct rec_type *prec;
char *rname;

int cond;

#endif DEBUG

printf("e rt_search\n");
#endif

if (prec == NULL)
return(NULL);
else if ((cond = strcmp(rname, prec->name)) == MATCH)
return(prec); /*name found - return pointer*/
else if (cond < 0) /*look in left sub tree*/
return(rt_search(prec->left, rname));
else /*look in right sub tree*/
return(rt_search(prec->right, rname));
}
file    line level source

rec_t  200  0  /*Rt_write saves information on the record types by writing it to the DFM.*/
rec_t  201  0
rec_t  202  0
rec_t  203  0  rt_write(fp,prec,cnt)
rec_t  204  0  FILE *fp;
rec_t  205  0  struct rec_type *prec;
rec_t  206  0  int *cnt;
rec_t  207  0  {
    int i;
rec_t  208  1
rec_t  209  1
rec_t  210  1  #ifdef DEBUG
rec_t  211  1  printf("e rt_write\n");
rec_t  212  1  #endif
rec_t  213  1
rec_t  214  1  if (prec == NULL)
rec_t  215  1      return;
rec_t  216  1  else
rec_t  217  1    {
rec_t  218  2    if (prec->rc1.add_on)  /*flush buffer*/
rec_t  219  2      write_buf(prec->rc1.fd,prec->rc1.final_fo,prec->rc1.file_buf,
rec_t  220  2      prec->rc1.final_bo);
rec_t  221  2    fprintf(fp, "%s %c %d %d \n", prec->name,prec->fdlim,prec->nosfield, prec->noskeyf);
rec_t  222  2    for (i=0; i<prec->noskeyf; i++)
rec_t  223  2      fprintf(fp,"%d ",*(prec->keyf+i));
rec_t  224  2    fprintf(fp, "%d %d\n", prec->rc1.final_fo, prec->rc1.final_bo);
rec_t  225  2    *cnt += 1;
rec_t  226  2  rt_write (fp,prec->left,cnt);
rec_t  227  2  rt_write (fp,prec->right,cnt);
rec_t  228  2  return;
rec_t  229  2  }

This file contains the routines to build the set type tree, insert a set type into the tree, and search the tree for a particular set type. A routine to write the information in the set type tree out to the DBM file is included.

/*MAD 10-81*/

/*Bset_tree reads the data base management file and builds a tree containing the information on the different set types in the data base*/

bset_tree (fp, number)

int i;

#define DEBUG
printf("e bset_tree\n");

#endif

for (i=1;i<=number; i++)

if (aset_tt (fp) == ERROR)
    Return(ERROR);

return(OK);
```c
/*A set tt will scan the dbm file and build a set type and place it in the
set type b-tree. Returns -1 if error occurs*/

set tt (fp) FILE *fp;

int 1, cond;
char owner[MAX_TNAME + 1];
char name[MAX_TNAME + 1];
extern struct set_type *st_root;
extern struct rec_type *rt_root;
struct set_type s;
struct set_type *st_insert();
struct rec_type *rt_search();

#define DEBUG

printf("e set tt\n");
#endif

/*read information for one set type*/

fscanf(fp,"%s %s %d %d\n",s.name,owner,name,member,&s.sci.final_go,
&st.root);
if ((s.owner = rt_search(rt_root,owner)) == NULL ||
(s.member = rt_search(rt_root,member)) == NULL)
    return(error("unable to build set type - owner/member not a record type"));
if ((s.sci.fd = openfile(s.name, ST)) == ERROR ||
(s.sbtfd = openfile(s.name, BT)) == ERROR)
    return(OK);
else
    st.root = st_insert(st_root, &s, &cond);
    if (cond == MATCH)
        return(error("unable to build set type tree"));
    else
        return(OK);
```
/*st_insert inserts a new set type into the set type b-tree. If already
   present, condition is set to match(0). A pointer to the root is returned*/

struct set_type *st_insert(pset, s, cond)
struct set_type *pset;
struct set_type *s;
int *cond;
{
    long initbt();
    if (pset == NULL)
        /*build and add a new set type*/
        pset = STALLOC;
        strcpy(pset->sname,s->sname);
        pset->owner = s->owner;
        pset->member = s->member;
        *pset->cos = NULL;
        pset->scl.fd = s->scl.fd;
        pset->sbf.fd = s->sbf.fd;
        pset->sbfroot = initbt(s->sbf.fd);
        pset->scl.log_fd = 0;
        pset->scl.phy_fd = 0;
        pset->scl.final_fd = s->scl.final_fd;
        pset->scl.log_bo = 0;
        pset->scl.phy_bo = 0;
        pset->scl.final_bo = s->scl.final_bo;
        pset->scl.add_on = 0;
        pset->sleft = NULL;
        pset->sright = NULL;
        *cond = NO_MATCH;
    else if (((*cond = strcmp(s->sname,pset->sname)) == MATCH)
        error("set type already defined");
    else if (*cond < 0)       /*move left to place set*/
        pset->sleft = st_insert(pset->sleft, s, cond);
    else                    /*move right to place set*/
        pset->sright = st_insert(pset->sright, s, cond);
    return(pset);
/*Sett_add will add a set type to the data base.*/

`sett_add(stname,owner,member)`

```c
char *stname, *owner, *member;
```

```c
{ 
   extern struct rec_type *rt_root;
   extern struct set_type *st_root;
   struct set_type *st_search();
   extern cnt_st;
   struct set_type s;
   struct set_type *st_insert();
   struct rec_type *rt_search();
   int cond,fd;
   char *concat();
   char newn[MAX_TNAME + APN + 1];
   int verbose=NONE;

   #ifdef DEBUG
   printf("e sett_add \n");
   #endif

   if ((s.owner = rt_search(rt_root,owner)) == NULL) 
      (s.member = rt_search(rt_root,member)) == NULL)
      return(error("unable to build set type - owner/member not a record type");
   if (st_search(st_root,stname) != NULL)
      return(error("set type already defined");
   if ((s.sci.fd = create_file(stname, ST)) == ERROR)
      return(ERROR);
   concat(newn,stname,ST);
   if ((s.sbfid = creat(newn,PMODE)) ==ERROR)
      return(error("cannot create the set btree");
   buildfl(s.sbfid,SIZE,verbose);
   close(s.sbfid);
   if ((s.sbfid = open(newn,RW)) == ERROR)
      return(error("cannot open the set btree");
   else
      {
         strcpy(s.name, stname);
         s.sci.final_fd = 0;
         s.sci.final_bo = 0;
         st_root = st_insert(st_root, &s, Scond);
         if (cond == MATCH)
            return(ERROR);
         else
            {
               cnt_st +=1;
               return(OK);
            }
   }
}```
<table>
<thead>
<tr>
<th>file</th>
<th>line level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_t</td>
<td>170</td>
<td>1</td>
</tr>
<tr>
<td>set_t</td>
<td>171</td>
<td>0</td>
</tr>
<tr>
<td>set_t</td>
<td>172</td>
<td>0</td>
</tr>
<tr>
<td>set_t</td>
<td>173</td>
<td>0</td>
</tr>
<tr>
<td>set_t</td>
<td>174</td>
<td>0</td>
</tr>
<tr>
<td>set_t</td>
<td>175</td>
<td>0</td>
</tr>
</tbody>
</table>
set_t  Mon Sep 27 14:43:58 1982       Page 6

file    line level  source

set_t  177  0  /\*St_search searches the set type b-tree for a specific occurrence by
set_t  178  0     set type name. If present, a pointer to the set type is returned.  
set_t  179  0     Otherwise, null is returned*/ 
set_t  180  0 
set_t  181  0     struct set_type *st_search(pset,name) 
set_t  182  0     struct set_type *pset; 
set_t  183  0     char *name; 
set_t  184  0     { 
set_t  185  1     int cond; 
set_t  186  1     \#ifdef DEBUG 
set_t  187  1     printf("e st_search\n"); 
set_t  188  1     \#endif 
set_t  189  1 
set_t  190  1     if (pset == NULL) 
set_t  191  1     \*set not in tree*/ 
set_t  192  1     return(NULL); 
set_t  193  1     else if ((cond = strcmp(name,pset->sname)) == MATCH) 
set_t  194  1     \*match found*/ 
set_t  195  1     return(pset); 
set_t  196  1     else if (cond < @) 
set_t  197  1     \*look left*/ 
set_t  198  1     return(st_search(pset->sleft,name)); 
set_t  199  1     else 
set_t  200  1     \*look right*/ 
set_t  201  1     return(st_search(pset->sright,name)); 
set_t  202  0 
set_t  203  0 

/*Write_st saves information on the set types by writing it to the DBMP.*/

struct set_type *pset;
int *cnt;

int i;
struct rec_type *p;
char *owner;
char *member;

#ifndef DEBUG
    printf("write_st\n");
#endif

if (pset == NULL)
    return;
else
{
    if (pset->sci.add_on) /*flush buffer*/
        write_buf(pset->sci.fd,pset->sci.final_fo,
                  pset->sci.file_buf,pset->sci.final_bo);
    p = pset->owner;
close(pset->sobuf);
    owner = p->name; /*get name of owner record type*/
p = pset->member;
    member = p->name; /*get name of member record type*/
    fprintf(fp, "%s %s %d %d\n",pset->sname,owner,member,
            pset->sci.final_fo,pset->sci.final_bo);
    *cnt +=1;
    st_write(fp,pset->sleft,cnt);
    st_write(fp,pset->sright,cnt);
    return;
file

line level source

/*This file contains the routines to initialize the data base management system
and to store information on record and set types when use of system concludes.*/

/*MAD 10-81*/

/*Init opens the data base management file and sets up the record type b-tree
and the set type b-tree.*/

.

init()

FILE *fp, *fopen();

extern int cnt_rt;

extern long rbtree;

extern int rbtree;

extern int cnt_st;

long initbt();

#endif DEBUG

printf("e init\n");

#endif

if ((fp = fopen(DBMF, "r")) == NULL)

{ printf("record and set types have not been defined\n");

return(OK);

}

fscanf(fp, \"%d \%d\", &cnt_rt, &cnt_st);

if (brec_tree(fp, cnt_rt) == ERROR)

return(ERROP);

if (bset_tree(fp, cnt_st) == ERROR)

return(ERROP);

fclose(fp);

if ((rbtree = open(RBT, RW)) == ERROR) /*open and set record btree*/

return(error(\"unable to open dbm.rec.bt\"));

rbtree = initbt(rbtree);

return(OK);

}
/*Store dbmf stores the information about record types and set types on the
 *data base management file.*/

store_dbmf()
{
  FILE *fp;
  int cnt;
  extern int cnt_rt;
  extern int cnt_st;
  extern struct rec_type *rt_root;
  extern struct set_type *st_root;
  extern int rbtfd;

  #ifdef DEBUG
  printf("e store_dbmf");
  #endif

  fp = fopen(DBMF, "w");
  fprintf(fp, "%d %d\n", cnt_rt, cnt_st);
  cnt = 0;
  rt_write(fp, rt_root, &cnt);
  if (cnt != cnt_rt)
    fprintf(stderr, "error during write of record tree on DBMF\n");
  cnt = 0;
  st_write(fp, st_root, &cnt);
  if (cnt != cnt_st)
    fprintf(stderr, "error during write of set tree on DBMF\n");
  fclose(fp);
  close(rbtfd);
  return;
}
utility 1 0 /*Utility contains functions that serve as utilities for the main program*/
utility 2 0
utility 3 0
utility 4 0
utility 5 0
utility 6 0 /*Error prints and error message and returns an error flag.*/
utility 7 0
utility 8 0 /*MAD 10/81*/
utility 9 0
utility 12 0 error(msg) /*error message*/
utility 11 0 char msg[];
utility 12 0 {
    fprintf(stderr,"%s\n",msg);
    return(ERROR);
}
utility 13 1
utility 14 1
utility 15 1
utility 16 0
utility 17 0
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>utility</td>
<td>19</td>
<td>0</td>
<td>/* Error prints an error message and the accompanying key and returns an error flag */</td>
</tr>
<tr>
<td>utility</td>
<td>20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>21</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>22</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>23</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>24</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>25</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>26</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>27</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>28</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>29</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>31</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>32</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>33</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>34</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

```c
error(msg, key)
char *msg;
char *key;
{
    fprintf(stderr, "%s Key = %s\n", msg, key);
    return(ERROR);
}
```
utility Mon Sep 27 14:44:19 1982

file | line | level | source
---|---|---|---
utility | 36 | 0 | /*Rec_str replaces the end of record mark with a null bit and therefore, in essence changes a record to a string.*/
utility | 37 | 0 |
utility | 38 | 0 | /*MAD 10-81*/
utility | 39 | 0 |
utility | 40 | 0 | rec_str(buf)
utility | 41 | 0 | char *buf;
utility | 42 | 0 | { int i;
utility | 43 | 1 | for (i=0; i< MAX_RECORD; i++)
utility | 44 | 1 | if(a(buf+i) == END_OF_RECORD)
utility | 45 | 1 | { *(buf+i) = NULL;
utility | 46 | 1 | return;
utility | 47 | 1 | }
utility | 48 | 2 | return(error("record too long"));
utility | 50 | 2 |
utility | 51 | 1 |
utility | 52 | 1 |
utility | 53 | 0 |
utility | 54 | 0 |
utility | 55 | 0 |
Utility Mon Sep 27 14:44:19 1982 Page 4

file line level source
utility 57 0 /*Str_rec replaces the null bit at end of string with an end of record mark
utility 58 0 and in essence changes a string to a record.*/
utility 59 0
utility 60 0 /*MAD 10-81*/
utility 61 0
utility 62 0 str_rec(buf)
utility 63 0 char *buf;
utility 64 0 {
    int i;
    for (i=0; i < MAX_RECORD; i++)
        if (*(buf+i) == NULL)
            *(buf+i) = END_OF_RECORD;
utility 67 1 return;
utility 68 1 }
utility 69 2 return(error("String too long"));
utility 70 2 }
utility 71 2
utility 72 1
utility 73 1
utility 74 0
utility 75 0
utility 76 0
utility 77 0
utility 78 0
```c
/* Amc copies one array of integers to another array.*/

utility 80 0
utility 81 0 /* MAD 10-81 */
utility 82 0
utility 83 0
utility 84 0
utility 85 0 int *top, *fromp;
utility 86 0 int size;
utility 87 0
utility 88 1
utility 89 1 for (i=0; i < size; i++)
utility 90 1 *top++ = *fromp++;
utility 91 1
utility 92 0
utility 93 0
utility 94 0
utility 95 0
utility 96 0
```
Utility will create and then open a file. The file opened will have a
name consisting of a name with a string appended to the end of it. */

/* MAD 10-21 */
create_file (file, append) char *filen, *append;

int fd;
char catfname[MAX_TNAME + APN + 1];
strcpy (catfname, filen);
strcat (catfname, append);
if ((fd = creat(catfname, PMODE)) == ERROR)
    return (error("unable to create file"));
else if ((fd = open(catfname, RW)) == ERROR)
    return (error("unable to open file"));
else
    return (fd);
<table>
<thead>
<tr>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
</table>
| 123  | utility | Openfile opens a character file for read/write options. The filename is a combination of a name and a string appended to the end of the name. If no error occurs, a file descriptor is returned. */
| 124  | utility | /*MAD 10-81*/
| 125  | utility | openfile (filen, append)
| 126  | utility | char *filen, *append;
| 127  | utility | int fd;
| 128  | utility | char catfname[MAX_TNAME + APN + 1];
| 129  | utility | strcpy (catfname, filen);
| 130  | utility | strcat (catfname, append);
| 131  | utility | if ((fd = open(catfname, RW)) == ERROR) return(error("unable to open file\n"));
| 132  | utility | else return(fd);
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>utility</td>
<td>147</td>
<td>0</td>
<td>/* Concatn concatenates two strings. */</td>
</tr>
<tr>
<td>utility</td>
<td>148</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>utility</td>
<td>149</td>
<td>0</td>
<td>char *concatn(newstring, bstring, estring)</td>
</tr>
<tr>
<td>utility</td>
<td>150</td>
<td>0</td>
<td>char *newstring,*bstring,*estring;</td>
</tr>
<tr>
<td>utility</td>
<td>151</td>
<td>0</td>
<td>{</td>
</tr>
<tr>
<td>utility</td>
<td>152</td>
<td>1</td>
<td>strcpy(newstring, bstring);</td>
</tr>
<tr>
<td>utility</td>
<td>153</td>
<td>1</td>
<td>strcat(newstring, estring);</td>
</tr>
<tr>
<td>utility</td>
<td>154</td>
<td>1</td>
<td>return(newstring);</td>
</tr>
<tr>
<td>utility</td>
<td>155</td>
<td>1</td>
<td>}</td>
</tr>
</tbody>
</table>
/*Read_rec reads a record from the file buffer into the record input/output
   buffer. */

/*MAD 10/81*/

read_rec(f_offset, b_offset, rec_to, pci)

long f_offset;        /* file offset of record to be read */
int b_offset;          /* block offset */
char *rec_to;          /* pointer to record input/output buffer */
struct cl *pci;       /* currency information for rec/set type */

/* log_fo, log_bo points at start of current of record type. 
   phy_fo, phy_bo points at end of current of record type. 
   final_fo, final_bo points at next place on file to add a 
   record. */

{ 
  int size;

  #ifdef DEBUG
  printf("entering read_rec\n");
  #endif

  if (f_offset < 0 || b_offset < 0 || pci->fd < 0)
    return(error("illegal input to read_rec");
  if (pci->phy_fo == 0L && pci->log_fo == 0L && pci->phy_bo == 0L &&
      pci->log_bo == 0L && pci->add_on == FALSE)
    {
      /* prime buffer the first time through */
      pci->phy_fo = pci->final_fo;
      pci->phy_bo = pci->final_bo;
      read_buf(pci->fd, pci->phy_fo, pci->file_buf, pci->phy_bo);
    }
  if (f_offset != pci->phy_fo && pci->add_on)
    { /* write to file previously added record */
      if (pci->final_bo == write_buf(pci->fd, pci->file_buf, 
          pci->final_fo, pci->final_bo) == FALSE)
        return(error("write incomplete");
      else
        return(error("write incomplete");
    }
  else
    return(error("write incomplete");
  pci->log_fo = f_offset;
  if (f_offset != pci->phy_fo)
    { /* reset currency pointers */
      pci->phy_fo = f_offset;
      if (pci->phy_fo != pci->final_fo)
        size = BLOCKSIZE;
      else
        size = pci->final_bo;
      if (read_buf(pci->fd, pci->phy_fo, pci->file_buf, size) == ERROR)
        return(error("unable to read input file properly");
  }
}
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>51</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>53</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>54</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>55</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>57</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>59</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>61</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

if (pci->add_on && b_offset >= pci->final_bo && (f_offset == pci->final_fo && b_offs
   return(error("unable to read input-record never written"));
   pci->log_bo = pci->phy_bo = b_offset;
   return(copy_in(rec_io, rec_io+MAX_RECORD, pci));

...
copy_in(rec_io,max_io,pci)
char *rec_io;    /* ... to record input/output buffer*/
char *max_io;    /* ... to 1st char beyond i/o buffer*/
struct ci *pci;  /*currency pointers for rec/set type*/

{    /*move record to i/o buffer*/
    /*number of characters to read*/

    #ifdef DEBUG
    printf("e copy_in\n");
    #endif

    while (pci->phy_bo < BLOCKSIZE &&
        (*rec_io++ = *(pci->file_buf+(pci->phy_bo))) != END_OF_RECORD &&
        rec_io < max_io)
    {    /*need to read next block from file*/
        pci->phy_bo += BLOCKSIZE;
        pci->phy_bo = 0;
        if(pci->phy_bo != pci->final_bo)
            size = BLOCKSIZE;
        else
            size = pci->final_bo;
        if (read_buf(pci->fd,pci->phy_bo,pci->file_buf,size) != ERROR)
            return(copy_in(rec_io,max_io,pci));
        else
            return(error("read into buffer incomplete"));
    }
}

else
    return(OK);
}
file line level source
10 109 0 /*Read_buf reads one block from a file into a buffer*/
10 110 0 /*MAD 10/81*/
10 111 0
10 112 0
10 113 0 read_buf (fd, offset, file_buf, size)
10 114 0 int fd;
10 115 0 /*file descriptor for input file*/
10 116 0 long offset; /*offset of desired block of file*/
10 117 0 char *file_buf; /*file input buffer*/
10 118 0 int size; /*number of characters to read*/
10 119 0{
10 120 1 #ifdef DEBUG
10 121 1 char *t;
10 122 1 printf( "read=\n");
10 123 1 for (i=file_buf; i<file_buf+size; i++)
10 124 1 printf("%c",*i);
10 125 1 printf("\n");
10 126 1 #endif
10 127 1 lseek(fd, offset, START); /*move to offset in file*/
10 128 1 return(read(fd, file_buf, size));
10 129 1
10 130 1}
10 131 0
10 132 0
10 133 0
10 134 0
10 135 0
/*Write_rec writes one record from the input/output buffer to the file buffer*/

/*MAD 10/81*/

int write_rec (f_offset, b_offset, rec_io, pci)
{/*pointer to file offset of record to be written*/
  int *b_offset; /*pointer to block offset*/
  char *rec_io; /*pointer to record input/output buffer*/
  struct c1 *pci; /*currency information for rec/set type*/

  /*log_fo, log_bo points at start of current of record type.*/
  phy_fo, phy_bo points at end of current of record type.*/
  final_fo, final_bo points at next place on file to add
  a record.*/

  if (*f_offset == NEWL) /*add new record to file*/
  { /*final record not presently in core*/
    if (pci->log_fo == pci->phy_fo = *f_offset = pci->final_fo;
      pci->log_bo = pci->phy_bo = *b_offset = pci->final_bo;
      if (pci->add_on = TRUE;
        if (pci->phy_fo != pci->final_fo)
          cond = read_buf(pci->fd,pci->phy_fo,pci->file_buf,BLOCKSIZE);
        else
          cond = read_buf(pci->fd,pci->phy_fo,pci->file_buf,pci->final_bo);
        if (cond = ERROR;}
        else
          return(error("can not read output file"));
      copy_out(rec_io, rec_io+MAX_RECORD, pci);
      if (pci->add_on = TRUE;
        if (pci->final_fo = pci->phy_fo;
          pci->final_bo = pci->phy_bo;
        return(OK);
      }
    }
  }
  else
  { /*write over existing record*/
    if (pci->add_on & & pci->final_fo != *f_offset) /*write out new record left
    if (write_buf(pci->fd,pci->final_fo,pci->file_buf,pci->final_bo) = pci->fin
      return(error("file write improperly done");}
else
    pci->add_on = FALSE;
  
  if (pci->phy_fo != *f_offset)
    {
    if (*f_offset != pci->final_fo)
      cond = read_buf(pci->fd, f_offset, pci->file_buf, BLOCKSIZE);
    else
      cond = read_buf(pci->fd, f_offset, pci->file_buf, pci->final_bo);
    if (cond == ERROR)
      return(error("file read error"));
    }
  
  pci->log_fo = pci->phy_fo = *f_offset;
  pci->log_bo = pci->phy_bo = *b_offset;
  copy_out(rec_id, rec_nr + MAX_RECORD, pci);

  if (pci->phy_fo != pci->final_fo) /*write updated record*/
    cond = write_buf(pci->fd, pci->phy_fo, pci->file_buf, BLOCKSIZE);
  else
    cond = write_buf(pci->fd, pci->phy_fo, pci->file_buf, pci->final_bo);

  if (cond == BLOCKSIZE || cond == pci->final_bo)
    return (OK);
  else
    return(error("write not completed"));
file
line level source
217 /* Copy_out copies a record from an input/output area and places it in the file
218 buffer. If need be, the file buffer is written out and a new one is made
219 available*/
220
221 /* MAD 10/81*/
222
223 copy_out (rec_io,max_io,pci)
224 char *rec_io; /* " record input/output buffer */
225 char *max_io; /* " 1st character past record I/O buffer */
226 struct ci *pci; /* currency information for record/set type */
227 {
228    int cond;
229    char *i;
230
231 #ifdef DEBUG
232    printf("entering copy_out\n");
233 #endif
234
235 for (i=rec_io; i<max_io & *i != END_OF_RECORD; ++i)
236     /* error check record size */
237 if (i == max_io)
238    return(error("output record to large");)
239 while (pci->phy_bo < BLOCKSIZE &
240        *(pci->file_buf + (pci->phy_bo++) = *rec_io++) != END_OF_RECORD)
241 if (*(rec_io-1) == END_OF_RECORD & pci->phy_bo = BLOCKSIZE)
242 {
243      if (!pci->add_on) /* write revised record */
244          {
245              if (pci->phy_bo = pci->final_bo)
246                  if (BLOCKSIZE != write_buf(pci->fd,pci->phy_bo,pci->file_buf,BLOCKSIZE)
247                      return(OK);
248              else if (pci->final_bo = write_buf(pci->fd,pci->phy_bo,pci->file_buf,pci
249                      return(OK);
250              return(error("write not completed");)
251          }
252          else if (BLOCKSIZE = write_buf(pci->fd,pci->phy_bo,pci->file_buf,BLOCKSIZE)
253              /* write buffer area to output */
254              pci->phy_bo += BLOCKSIZE;
255              pci->phy_bo = 0;
256              if (pci->add_on)
257                  {
258                      pci->final_bo = pci->phy_bo;
259                      pci->final_bo = 0;
260                  }
261 else
262          {
263              if (pci->phy_bo != pci->final_bo)
<table>
<thead>
<tr>
<th>line</th>
<th>source</th>
<th>line</th>
<th>source</th>
<th>line</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>267</td>
<td>3</td>
<td>268</td>
<td>3</td>
<td>269</td>
<td>3</td>
</tr>
<tr>
<td>270</td>
<td>3</td>
<td>271</td>
<td>3</td>
<td>272</td>
<td>3</td>
</tr>
<tr>
<td>273</td>
<td>2</td>
<td>274</td>
<td>2</td>
<td>275</td>
<td>1</td>
</tr>
<tr>
<td>276</td>
<td>1</td>
<td>277</td>
<td>1</td>
<td>278</td>
<td>0</td>
</tr>
<tr>
<td>279</td>
<td>0</td>
<td>280</td>
<td>0</td>
<td>281</td>
<td>0</td>
</tr>
</tbody>
</table>

```c
cond = read_buf(pci->fd,pci->phy fo,pci->file_buf,BLOCKSIZE);
else
cond = read_buf(pci->fd,pci->phy fo,pci->file_buf,pci->final bo);
if (cond == ERROR)
    return(error("read incomplete"));
return(copy_out(rec io,max io,pci));
else
    return(error("incomplete write to output file"));
```
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>283</td>
<td>0</td>
<td>/<em>Write_buf writes one block onto a file from a buffer</em>/</td>
</tr>
<tr>
<td>10</td>
<td>284</td>
<td>0</td>
<td>/<em>MAD 10/81</em>/</td>
</tr>
<tr>
<td>10</td>
<td>285</td>
<td>0</td>
<td>write_buf (fd,offset,file_buf,size)</td>
</tr>
<tr>
<td>10</td>
<td>286</td>
<td>0</td>
<td>int fd; /<em>file descriptor for output file</em>/</td>
</tr>
<tr>
<td>10</td>
<td>287</td>
<td>0</td>
<td>int offset; /<em>offset of desired block on file</em>/</td>
</tr>
<tr>
<td>10</td>
<td>288</td>
<td>0</td>
<td>char *file_buf; /<em>buffer for file</em>/</td>
</tr>
<tr>
<td>10</td>
<td>289</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>290</td>
<td>0</td>
<td>{</td>
</tr>
<tr>
<td>10</td>
<td>291</td>
<td>1</td>
<td>ifdef DEBUG</td>
</tr>
<tr>
<td>10</td>
<td>292</td>
<td>1</td>
<td>char *s;</td>
</tr>
<tr>
<td>10</td>
<td>293</td>
<td>1</td>
<td>printf(&quot;write=\n&quot;);</td>
</tr>
<tr>
<td>10</td>
<td>294</td>
<td>1</td>
<td>for(i=file_buf; i&lt;file_buf+size; i++)</td>
</tr>
<tr>
<td>10</td>
<td>295</td>
<td>1</td>
<td>printf(&quot;%c&quot;,*i);</td>
</tr>
<tr>
<td>10</td>
<td>296</td>
<td>1</td>
<td>printf(&quot;\n&quot;);</td>
</tr>
<tr>
<td>10</td>
<td>297</td>
<td>1</td>
<td>}</td>
</tr>
<tr>
<td>10</td>
<td>298</td>
<td>1</td>
<td>#endif</td>
</tr>
<tr>
<td>10</td>
<td>299</td>
<td>1</td>
<td>lseek(fd,offset,START); /<em>move to correct position in file</em>/</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
<td>1</td>
<td>return(write(fd,file_buf,size));</td>
</tr>
<tr>
<td>10</td>
<td>301</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>302</td>
<td>1</td>
<td>}</td>
</tr>
</tbody>
</table>
/* These variables are used in the dbup routines. */
#include "ext_dec"
char iobuf[MAX_RECORD + 2];
char key[MAX_KEY + 1];
char fp[MAX_KEY + 1];
char bp[MAX_KEY + 1];
char savekey[MAX_KEY + 1];
char savenext[MAX_KEY + 1];
char unit[MAX_KEY + 1];
struct ptrs os, saveos, saveos2;
struct rec_type *rt;
struct set_type *st;
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbup</td>
<td>1</td>
<td>0</td>
<td>/<em>MAD 11/81</em>/</td>
</tr>
<tr>
<td>dbup</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>4</td>
<td>0</td>
<td>/<em>This file contains the following routines to update the database.</em>/</td>
</tr>
<tr>
<td>dbup</td>
<td>5</td>
<td>0</td>
<td>ra - add a record occurrence to the database - recadd</td>
</tr>
<tr>
<td>dbup</td>
<td>6</td>
<td>0</td>
<td>ao - add an owner set occurrence - ownadd</td>
</tr>
<tr>
<td>dbup</td>
<td>7</td>
<td>0</td>
<td>am - add a member set occurrence - mebadd</td>
</tr>
<tr>
<td>dbup</td>
<td>8</td>
<td>0</td>
<td>dr - delete a record occurrence from the database - recdel</td>
</tr>
<tr>
<td>dbup</td>
<td>9</td>
<td>0</td>
<td>do - delete an owner set occurrence - owndel</td>
</tr>
<tr>
<td>dbup</td>
<td>10</td>
<td>0</td>
<td>dm - delete a member set occurrence -.mebdel</td>
</tr>
<tr>
<td>dbup</td>
<td>11</td>
<td>0</td>
<td>fr - find a record - recfind</td>
</tr>
<tr>
<td>dbup</td>
<td>12</td>
<td>0</td>
<td>fo - find an owner record of a specific member in a set - ownfind</td>
</tr>
<tr>
<td>dbup</td>
<td>13</td>
<td>0</td>
<td>ff - find the first member of an owner in a set - mebfind</td>
</tr>
<tr>
<td>dbup</td>
<td>14</td>
<td>0</td>
<td>fn - find next owner or member in a set - nextfind</td>
</tr>
<tr>
<td>dbup</td>
<td>15</td>
<td>0</td>
<td>co - change owner of specific member to new owner in same set - cnewon</td>
</tr>
<tr>
<td>dbup</td>
<td>16</td>
<td>0</td>
<td>ca - change all members of an owner to new owner in same set - cngall</td>
</tr>
<tr>
<td>dbup</td>
<td>17</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>18</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>19</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>21</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>22</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>23</td>
<td>0</td>
<td>setoadd</td>
</tr>
<tr>
<td>dbup</td>
<td>24</td>
<td>0</td>
<td>owntdel</td>
</tr>
<tr>
<td>dbup</td>
<td>25</td>
<td>0</td>
<td>mebtdel</td>
</tr>
<tr>
<td>dbup</td>
<td>26</td>
<td>0</td>
<td>rscor</td>
</tr>
<tr>
<td>dbup</td>
<td>27</td>
<td>0</td>
<td>savecor</td>
</tr>
<tr>
<td>dbup</td>
<td>28</td>
<td>0</td>
<td>putpts</td>
</tr>
<tr>
<td>dbup</td>
<td>29</td>
<td>0</td>
<td>getpts</td>
</tr>
<tr>
<td>dbup</td>
<td>30</td>
<td>0</td>
<td>valkr</td>
</tr>
<tr>
<td>dbup</td>
<td>31</td>
<td>0</td>
<td>valks</td>
</tr>
<tr>
<td>dbup</td>
<td>32</td>
<td>0</td>
<td>bldkey</td>
</tr>
<tr>
<td>dbup</td>
<td>33</td>
<td>0</td>
<td>nfields</td>
</tr>
<tr>
<td>dbup</td>
<td>34</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>35</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>36</td>
<td>0</td>
<td>NOTE: dbup_c_dec includes declarations that are common to the update routines and includes declarations of all external variables.*/</td>
</tr>
<tr>
<td>dbup</td>
<td>37</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>38</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>39</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
/*Recadd will add records to a record file type as designated by the c...*/
recadd (ifp)
FILE *ifp;
{
#include "dbup.c_dec"
char c;
int more;
char *i;
struct set_type *pst;
int n;

#define DEBUG
printf ("entering record add\n");
#endif

more = TRUE;
while (more)
{
  if ((c = getc(ifp)) == EOF)
    more = FALSE;
  else
    {
      ungetc(c,ifp);
      i = 1obuf;
      while ((c = getc(ifp)) != END_OF_RECORD &&
      (i < 1obuf + MAX_RECORD))
        *i++ = c;
      if (1 == 1obuf + MAX_RECORD)
        return(error("input record too long");
      *i++ = END_OF_RECORD;
      *i = NULL;
    /*validate input record*/
      if((n=nfields(1obuf,cor_rt->fdlim)) != cor_rt->nsofield)
        return(error("illegal # of fields for record type");
      else if (bldkey(1obuf,key) == ERROR)
        return(ERROR);
      else if (item_search(rbifd,rbroot,key,6os) == PRESENT)
        printf(stderr,"%s duplicate key\n", key);
      else
        {  /*write rec to file and save offsets*/

          #ifdef DEBUG
          printf("key= %s\n",key);
          #endif
          os.btd = NEWL;
os.btb = NEW;
if (write_rec(&os.btd,&os.btb,1obuf,&cor_rt->rc1) == ERROR)
    return(ERR); 
else if (item_insert(rbtd,&rbtree,key,&os) == NULL)
    return(ERR);
else
{
    strcpy(cor_u1nt, key);
    pst = st_root;
    set_add (pst); /*add key as owner in sets*/
}
return(OK);
/*Ownadd adds an owner set occurrence to a set type*/

void ownadd()
{
    #include "dbup_c_dec"

    #ifdef DEBUG
    printf("e ownadd\n");
    #endif

cor_rt = cor_st->owner;
    if (valkr(os, iobuf) != ERROR) /*validate key and record format*/
    {
        os.btd = NEWL; /*build new owner record occur*/
        os.btb = NEW;
        *fp = NULL;
        *bp = NULL;

        /*store the new owner record and offset*/
        if (putpts(fp, bp, &os) == ERROR ||
            item_insert(cor_st->sbfld, &cor_st->sbtroot, cor_unit, &os)
                ? PRESENT)
                return(ERROR);

        return(OK);
    }

    return(ERROR); /*invalid key or record format*/

}
/*Mebadd adds member set occurrence to a set type*/

mebadd(mkey)
char *mkey;
{
#include "dbup_c_dec"

ifdef DEBUG
printf("mebadd
");
endif

cor_rt = cor_st->member;
strcpy(savekey, cor_unit); /*save owner key*/
strcpy(cor_unit, mkey); /*set currency to member key*/
if (valkr(Sos, iobuf) == ERROR)
{
    strcpy(cor_unit, savekey); /*invalid key or record format*/
    return(OK);
}
else if (item_search(cor_st->sbtfd, cor_st->sbtroot, savekey, &sos)
   != PRESENT) /*make sure owner is in set type*/
{
    strcpy(cor_unit, savekey);
    return(error("owner not a member of settype", cor_unit));
}
else if (getpts(fp, bp, &sos) == ERROR) /*get owner pointers*/
    return(HELPER);
else if (*bp != NULL)
    return(error("owner key is member in the set"));
else
{
    /*set owner forward pointer to new member*/
    strcpy(savenext, fp);
    strcpy(fp, cor_unit);
    putpts(fp, bp, &sos);
    if (*savenext != NULL) /*build a new member record*/
        strcpy(fp, savekey);
    else
        strcpy(fp, savenext);
    strcpy(bp, savekey);
    os.btd = NEWL;
    os.btt = NEW;
    putpts(bp, bp, &sos); /*store new member occr rec*/
    item_insert(cor_st->sbtfd, &cor_st->sbtroot, cor_unit, &sos);
    if (*savenext != NULL) /*fix backward pointer of next member*/
    {
        if (item_search(cor_st->sbtfd, cor_st->sbtroot, savenext, 
            Sos) != PRESENT ||
            getpts(fp, bp, &sos) == ERROR ||
            strcmp(bp, savekey) != MATCH)
file     line  level     source

dbup   200   3
dbup   201   3
dbup   202   3
dbup   203   3
dbup   204   3
dbup   205   2
dbup   206   1
dbup   207   0
dbup   208   0
dbup   209   0
dbup   210   0
dbup   211   0

return(kerror("set processing error",savenv));
strpy(bp,cor_unit);
putps(fp, bp, Sos);
return(0);
/* Recdel deletes a record from the database record file. Any set occurrences for this key are also deleted. */

recdel()
{
    #include "dbup_c_dec"
    struct set_type *pst;
    #ifdef DEBUG
    printf("entering recdel\n");
    #endif
    if (item_delete(rbtfd, &rbtroot, cor_unit,&os) != PRESENT)
        return(NOTPRESENT);
    pst = st_root; /* now delete all set occur*/
    ownerdel(pst); /*delete owner occur*/
    pst = st_root;
    memberdel(pst); /*delete member occur*/
    return(PRESENT);
}
/*owndel deletes an owner occurrence in a set. This causes the record in the
record file to be deleted and all members to be deleted. Also, any set that
contains this owner as a member is also visited and the owner-now-member
is deleted.*/

owndel()
{
    #include "dbup_c_def"
    int more;
    #ifndef DEBUG
    printf("e owndel\n");
    #endif
    cor_rt = cor_st->owner;
    if (item_delete(cor_st->sbtfd,&cor_st->sbtroot,cor_unit,&os) != PRESENT)
        return(NOTPRESENT);
    else
    {
        #ifndef DEBUG
        printf("after delete fo= %d, bo= %d\n",os.btd,os.btb);
        #endif
        savecor(unit,&st,&rt);    /*save currency pointers*/
    }
    #ifndef DEBUG
    printf("after savecor- unit=%s, st= %d, rt= %d\n",unit,st,rt);
    printf("should be -coru= %s, st= %d, rt= %d\n",cor_unit,cor_st,cor_rt);
    #endif
    more = TRUE;
    while (more)
    {    /*get key for next member*/
        getpts(fp,bp,&os);
        #ifndef DEBUG
        printf("getpts- fp= %s, bp= %s\n",fp,bp);
        #endif
        if (*fp == NULL)
            more = FALSE;
        else
        {
            strcpy(cor_unit,fp);    /*set currency for next member*/
            memdel(os.btd,os.btb);
        }
    }
    rsCor(unit,&st,&rt);    /*reset currency*/
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbup</td>
<td>290</td>
<td>1</td>
<td>ifdef DEBUG</td>
</tr>
<tr>
<td>dbup</td>
<td>291</td>
<td>1</td>
<td>printf(&quot;after rscor= coru= %s, corst= %d, corrt= %d\n&quot;, cor_unit, cor_st, cor_rt);</td>
</tr>
<tr>
<td>dbup</td>
<td>292</td>
<td>1</td>
<td>endif</td>
</tr>
<tr>
<td>dbup</td>
<td>293</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>294</td>
<td>1</td>
<td>recdel(); /<em>delete the record from the record ttree</em>/</td>
</tr>
<tr>
<td>dbup</td>
<td>295</td>
<td>1</td>
<td>rscor(unit,&amp;st,&amp;rt);</td>
</tr>
<tr>
<td>dbup</td>
<td>296</td>
<td>1</td>
<td>return(OK);</td>
</tr>
<tr>
<td>dbup</td>
<td>297</td>
<td>1</td>
<td>}</td>
</tr>
<tr>
<td>dbup</td>
<td>298</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>299</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>300</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>301</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>302</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```c
/* mebdel deletes a member from a set occurrence. If the prior member or owner
   is not in the btree for the set, then the file offset and block offset passed
   to the routine as parameters are checked. If the fos is equal to newl, the routine is
   in the middle of a recursive call and this member will be deleted later. If
   fos is not equal to newl, it then contains the file offset of the prior member
   and is used to get the prior member's pointers.*/

#define DEBUG

int fos; int bos;

#include "dbup_c_dec"

mebdel(fos, bos)

if (item_delete(cor_st->sbtfd, &cor_st->sbtroot, cor_unit, &os) != PRESENT)
    return(NOTPRESENT);

saveos = os; os = fos;

getpts(fp, bp, &os);
if (*bp == NULL || *fp == NULL)
    return(kerror("delete processing error", cor_unit));

.GetItem pointers for this member*/

if (item_search(cor_st->sbtfd, cor_st->sbtroot, bp, &os) != PRESENT)
    if (fos == NEW)
        return(NOTPRESENT); /* will get this member later*/
    else
        os.btd = fos;
        os.btb = bos;

printf("back from i_search fo= %ld, bo= %d\n", os.btd, os.btb);
#endif

strcpy(savekey, bp);
strcpy(savenext, fp);

#define DEBUG

printf("savekey= %s, savenext= %s\n", savekey, savenext);
#endif

getpts(fp, bp, &os); /* get prior key's record*/
if (strcmp(savekey, savenext) == "MATCH")
```
```c
{ /*last member for this owner*/
    if (bp != NULL)
        return(kerror("delete processing error", bp));
    else
    {
        *fp = NULL;
        putpts(fp, bp, &os); /*reset prior's pointers*/
    }
}
else
{
    strcpy(fp, savename); /*reset prior's pointers*/
    putpts(fp, bp, &os);
    if (item_search(cor_st->sbf1, cor_st->sbfroot, savename, &os)
        != PRESENT)
        return(kerror("missing set links", savename));
    retpts(fp, bp, &os); /*reset next member's pointers*/
    if (*bp != NULL)
        strcpy(bp, savekey);
    putpts(fp, bp, &os);
}
rscor(unit, &st, &rt); /*now delete the record if present*/
recdel(); /*now delete the record if present*/
return(OK);
```
/*Find outputs a record using the currency pointers to decide which key*/

recfind (ofp)

FILE *ofp;

#include "dbup_c.dec"

char c, &cptr;

/*get the record from db storage*/

#define DEBUG

printf( "entering recfind\n" );

if (item_search(rbtd,rbtree,cor_unit,&os) != PRESENT)
  return(kerror(rbtd,rbtree,cor_unit,&os));

/*get record into io buffer*/

if (read_rec(os,btd,os.btb,iobuf,&cor_rt->rci) == ERROR)
  return (ERROR);

  c = 'p';
  cptr = iobuf;
  while (c != END_OF_RECORD)
    {
      c = *cptr++;
      putc (c, ofp);
    }

return(OK);
/* Ownfind finds the owner for a member key and outputs the record for that owner. */

FILE *ofp;
#include "dbup_c_dec"

/* ifdef DEBUG */
printf("e ownfind\n");
/* endif */

/* find set owner occur */
cor_rt = cor_st->owner;
if (item_search(cor_st->sbtfd,cor_st->sbtroot,cor_unit,os) != PRESENT)
   return(kerror("key absent from set type",cor_unit));
getpts(fp,bp,os);
if (*bp == NULL)
   return(kerror("key is an owner",cor_unit));
while (*bp != NULL)
   { if (item_search(cor_st->sbtfd,cor_st->sbtroot,fp,os) != PRESENT)
      return(kerror("processing error-missing set link",cor_unit));
      strcpy(cor_unit,fp);
      getpts(fp,bp,os);
   }
recfind(ofp); /* output owner record */
return(0);
/*Mebfind finds the first member record for an owner*/

#include "dbup_c_defs"

#define DEBUG

printf("e mebfind \n");

#endif

if (item_search(cor_st->sbtfd, cor_st->sbroot, cor_unit, &os) != PRESENT)
    return(kerror("key absent from set type", cor_unit));

getpts(fp, bp, &os);

if (*bp != NULL)
    return(kerror("key not an owner", cor_unit));

if (*fp == NULL)
    return(kerror("key has no members", cor_unit));

strcpy(cor_unit, fp); /*reset currency to member's record*/

strcpy(cor_st->cos, cor_unit);

recfnd(ofp); /*output record*/

return(OK);
file line level source

dbup 488 0 /*Nextfind outputs the record of the next key in the set link*/
dbup 489 0

dbup 490 0 nextfind (ofp)
dbup 491 0 FILE *ofp;
dbup 492 0 {
  #include "dbup_c_dec"

#define DEBUG

dbup 496 1
  printf("e nextfind\n");
dbup 497 1
  #endif

dbup 498 1

if ((*cor_st->cos) == NULL)
  return(error("undefined current of set type"));

dbup 502 1
  strcpy(cor_unit,cor_st->cos);

if(item_search(cor_st->sbtfi,cor_st->sbtroot,cor_unit,&os) != PRESENT)
  return(error("key absent from set type",cor_unit));

dbup 505 1
  getpts(fp,bp,&os); /*find next key in set*/

if (*fp != NULL & & bp != NULL)
{
  strcpy(cor_unit,fp);
  if (item_search(cor_st->sbtfi,cor_st->sbtroot,cor_unit,&os) != PRESENT)
    return(error("missing set link",cor_unit));

  getpts(fp,bp,&os); /*set cor record type*/

  dbup 512 2
  if (*bp == NULL)
    return (error ("last member reached"));

  else
    {
      cor_rt = cor_st->member;
      strcpy(cor_st->cos, cor_unit); /*reset cur set type*/
      recfind (ofp); /*output record*/
      return (OK);
    }

  } else
    return(error("owner key is current of set type"));

dbup 525 0


file    line  level  source

dbup   531   0  /* Cnghown changes a member in a set occurrence to a new owner in the same set */
dbup   532   0

dbup   533   0
dbup   534   0
cnghown(newokey)
dbup   535   0
char *newokey;
dbup   536   0

#include "dbup_c_dec"
dbup   537   1
#endif

(dbup   538   1

printf("cnghown\n");
#endif

dbup   539   1

/* validate new owner key and member key */
dbup   540   1
if (valks(newokey,&os,&saveos) == ERROR)
dbup   541   1
    return(ERROR);

(dbu   542   1
    getpts(fp,bp,&os);
    /* set member's links*/
/dbu   543   1
    if (*bp == NULL)
/dbu   544   1
        return(kerror("key is owner in this set",cor_unit));
/dbu   545   1
    if ((item_search(cor_st->sbfid,cor_st->sbrt,fp,&saveos2) != PRESENT)
/dbu   546   1
        return(kerror("missing set links",fp));
/dbu   547   1
    if (strcmp(fp, bp) == MATCH)
/dbu   548   1
        {  /* final member in this occur.*/
/dbu   549   2
            getpts(fp,bp,&saveos2);
/dbu   550   2
            *fp = NULL;
/dbu   551   2
            putpts(fp,bp,&saveos2);
/dbu   552   2
        }
/dbu   553   1
    else
/dbu   554   1
        {
/dbu   555   2
            strcpy(savenext,fp);
/dbu   556   2
            strcpy(savekey,bp);
/dbu   557   2
            getpts(fp, bp, &saveos2);
/dbu   558   2
            strcpy(fp, savenext);
/dbu   559   2
            putpts(fp, bp, &saveos2);
/dbu   560   2
            if ((item_search(cor_st->sbfid,cor_st->sbrt, savenext, &saveos2) != PRESENT)
/dbu   561   2
                return(kerror("missing set link", savenext));
/dbu   562   2
            getpts(fp, bp, &saveos2);
/dbu   563   2
            if (*bp != NULL)
/dbu   564   2
                strcpy(bp, savekey);
/dbu   565   2
            putpts(fp, bp, &saveos2);
/dbu   566   2
        }
/dbu   567   1
    /* now add member to new owner*/
/dbu   568   1
    getpts(fp,bp,&saveos);
/dbu   569   1
    strcpy(savenext,fp);
/dbu   570   1
    strcpy(fp,cor_unit);
/dbu   571   1
    putpts(fp,bp,&saveos);
/dbu   572   1
    if (*savenext != NULL)
/dbu   573   1
        
/dbu   574   1
    getpts(fp,bp,&saveos);
/dbu   575   1
    strcpy(savenext,fp);
/dbu   576   1
    strcpy(fp,cor_unit);
/dbu   577   1
    putpts(fp,bp,&saveos);
/dbu   578   1
    if (*savenext != NULL)
/dbu   579   1
        
/dbu   580   1
if(item_search(cor_st->sbtree, cor_st->sbtree, saveneveext, &saveos) != PRESENT)
    return(kerror("missing links", saveneveext));
getpts(fp, bp, &saveos);
strcpy(savekey, bp);
strcpy(bp, cor_unit);
putpts(fp, bp, &saveos);
}
else
    strcpy(saveneext, newokey); /*new owner has no members*/
strcpy(fp, saveneext); /*set member's new links*/
strcpy(bp, newokey);
putpts(fp, bp, &os);
return(OK);
/* Cngall changes all the members in a set occurrence to a new owner within
the same set. */
cngall(newokey)
char *newokey;

#include "dbup_c_dec"

if (valks(newokey,&os,&saveos) == ERROR)
    return(ERROR);
getpts(fp,bp,&os); / *get member key*/
while (*fp != NULL)
{
    strcpy(cor_unit,fp); / *move each member to newowner*/
    if (cngown(newokey) == ERROR)
        return(ERROR);
    getpts(fp,bp,&os);
}
strcpy(cor_unit,newokey);
/* Service routines */

/* Setoadd searches the set type binary tree and if a match of owner and
  cor record type is found, adds the new record key as an owner to the set*/

setoadd(pst)
struct set_type *pst;
{
  #include "ext_dec"
  #ifdef DEBUG
  printf("e setoadd\n");
  #endif
  if(pst == NULL)
    return;
  else
  {
    if (pst->owner == cor_rt)
      {
        cor_st = pst;
        ownadd();
      }
    setoadd(pst->left);
    setoadd(pst->sright);
  }
/* Owntdel searches the set type binary tree and if a match of owner and cor record type is found, deletes any owner record with cor unit as key. */

struct set_type *pst;

#include "ext_dec"

ifdef DEBUG
  printf("e owntdel\n");
endif

if (pst == NULL)
  return;
else
  if (pst->owner == cor_rt)
    cor_st = pst;
    owndel();
  owndel(pst->sleft);
  owndel(pst->sright);
/**Mebtde1 searches the set type tree and if a match of member and cor_rt is found, deletes the owner record of cor_unit.**

```
mebtde1(pst)
struct set_type *pst;
{
    #include "ext_dec"
    #ifndef NDEBUG
    printf("e mebtde1\n");
    
    if (pst == NULL)
        return;
    else
        {
            if (pst->member == cor_rt)
            {
                cor_st = pst;
                mebtde1(NEWL,NEW);
            }
            mebtde1(pst->sleft);
            mebtde1(pst->sright);
            return;
        }
```
file: rscor resets the currency pointers

/* */

struct rec_type **rt;
struct set_type **st;
{
    #include "ext_dec"

    #ifdef DEBUG
    printf( "e rscor\n" );
    #endif
    strcpy(cor_unit,unit); cor_st = *st; cor_rt = *rt;
    return;

    }
<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>level</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbup</td>
<td>755</td>
<td>0</td>
<td>/<em>Savecor saves the currency pointers</em>/</td>
</tr>
<tr>
<td>dbup</td>
<td>756</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>757</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>758</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>759</td>
<td>0</td>
<td>char *unit;</td>
</tr>
<tr>
<td>dbup</td>
<td>760</td>
<td>0</td>
<td>struct set_type **st;</td>
</tr>
<tr>
<td>dbup</td>
<td>761</td>
<td>0</td>
<td>struct rec_type **rt;</td>
</tr>
<tr>
<td>dbup</td>
<td>762</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>763</td>
<td>1</td>
<td>#include &quot;ext_dec&quot;</td>
</tr>
<tr>
<td>dbup</td>
<td>764</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>765</td>
<td>1</td>
<td>#ifdef DEBUG</td>
</tr>
<tr>
<td>dbup</td>
<td>766</td>
<td>1</td>
<td>printf(&quot;e savecor\n&quot;);</td>
</tr>
<tr>
<td>dbup</td>
<td>767</td>
<td>1</td>
<td>#endif</td>
</tr>
<tr>
<td>dbup</td>
<td>768</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>769</td>
<td>1</td>
<td>strcpy(unit, cor_unit);</td>
</tr>
<tr>
<td>dbup</td>
<td>770</td>
<td>1</td>
<td>*st = cor_st;</td>
</tr>
<tr>
<td>dbup</td>
<td>771</td>
<td>1</td>
<td>*rt = cor_rt;</td>
</tr>
<tr>
<td>dbup</td>
<td>772</td>
<td>1</td>
<td>return;</td>
</tr>
<tr>
<td>dbup</td>
<td>773</td>
<td>1</td>
<td>}</td>
</tr>
<tr>
<td>dbup</td>
<td>774</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>775</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>776</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>777</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbup</td>
<td>778</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Putpts builds a set type record containing the links for an owner or member in a set occurrence. Once the record is built, it is output to the set file.*

```c
#include "ext_dec"

char *buf[MAX_RECORD+1];
char *pt;

#define DEBUG

#define fprintf("e putpts\n");

#define endif

pt = iobuf;
c = 'a';
while (c != NULL) {
    c = *pt++ = *fp++;
}
```

The code snippet above is a C program that builds a set type record containing links for an owner or member in a set occurrence. Once the record is built, it is output to the set file. The code includes the necessary include statements, declarations, and a function definition for `putpts`, which appears to be a part of a larger set management system.
/*Getpts retrieves the forward and rear pointers after reading a set type record*/

getpts (fp, bp, os)
    char *fp,*bp;
    struct ptrs *os;
{
    #include "ext_dec"
    char Iobuf[MAX_RECORD + 1];
    char *pt;
    #ifdef DEBUG
    printf("e getpts\n");
    #endif
    if(read_rec(os->btd,os->bth,1obuf,&cr_st->sci) == ERROR)
    return(OK);
    pt = Iobuf;
    while((*fp++ = *pt++) != NULL) /*get forward pointer*/
    ;
    pt = Iobuf + MAX_KEY + 1;
    while((*bp++ = *pt++) != NULL) /*get backward pointer*/
    ;
    return(OK);"
```c
/* Valkr validates the key for an add of owner or member into set type and
   checks record against cor record type format. */
valkr (os, iobuf)
struct pts *os;
char *iobuf;
{
#include "ext_dec"

char key[MAX_KEY + 1];
#endif DEBUG

printf("e valkr\n");
#endif

if(item_search(cor_st->sbtfd, cor_st->sbtroot, cor_unit, os) == PRESENT)
   return(kerror("key already present in set type", cor_unit));
#endif

if(item_search(rbtfd, rbtroot, cor_unit, os) != PRESENT)
   return(kerror("record not present in database", cor_unit));
#endif

if (read_rec(os->btb, os->btb, iobuf, &cor_rt->rc1) == ERROR)
   return(ERROR);
#endif

if (bldkey (iobuf, key) == ERROR || strcmp(key, cor_unit) != MATCH)
   return(kerror("record doesn't fit set type", key));
else
   return(OK);

return(OK);
}
/* Valks validates the keys input for change owner and change all commands */

vals(newkey, os, saveos)
char *newkey;
struct ptrs *os, *saveos;
{
#include "ext_dec"

#define DEBUG

printf("e vals\n");
#endif

if (item_search(cor_st->sbtd, cor_st=sbroot, newkey, saveos) != PRESENT)
return(errort("new owner key absent from set", newkey));
if (item_search(cor_st->sbtd, cor_st->sbroot, cor_unit, os) != PRESENT)
return(errort("key absent from set", cor_unit));
return(OK);
valchar (key) /*validate characters in key are printable*/
char *key; /*ascii dependent*/

ifdef DEBUG

printf ("e valchar\n");

while (*key != NULL) {
    if (*key < ' ' || *key > '~') {
        return(error("invalid character in key"));
    }
    else 
        key++;

return (OK);
}
file | line | level | source
--- | --- | --- | ---
dbup | 985 | 0 | /*nfields returns the number of fields in a string*/
dbup | 986 | 0 | 

dbup | 987 | 0 | 

dbup | 988 | 0 | nfields (string, fd1m)
dbup | 989 | 0 | char *string;
dbup | 990 | 0 | char fd1m; /*field delimiter*/
dbup | 991 | 0 | { int num; /*number of fields*/
dbup | 992 | 1 | 

dbup | 993 | 1 | 

dbup | 994 | 1 | #ifdef DEBUG

dbup | 995 | 1 | printf("e nfields, fd1m= %c
", fd1m);
dbup | 996 | 1 | #endif

dbup | 997 | 1 | num = 0;
dbup | 998 | 1 | 

dbup | 999 | 1 | for ( ; *string != NULL; string++)

dbup | 1000 | 1 | if ( *string == fd1m | *string == END_OF_RECORD)

dbup | 1001 | 1 | num += 1;

dbup | 1002 | 1 | return (num);

dbup | 1003 | 1 | }